NAWCWD TP 8525
NAVAIR

Implementing the Region Growing Method Part 1: The Piecewise Constant Case

by
Jorge M. Martin
Research Department

SEPTEMBER 2002

NAVAL AIR WARFARE CENTER WEAPONS DIVISION
CHINA LAKE, CA 93555-6100

Naval Air Warfare Center Weapons Division

FOREWORD

This report presents an implementation of the *Region Growing method* for minimizing the simplified Mumford and Shah functional in the two-dimensional, multichannel, piecewise constant setting. It describes in detail the Matlab functions used in the implementation and contains the listings of the programs. The work was done at the Naval Air Warfare Center Weapons Division from February 1996 to September 1997. It was funded by the NAWC In-House Laboratory Independent Research Program under Program Element 601152N, Work Unit Numbers 12304016 and A47D47, ILIR Project *Multi-Scale Segmentation Techniques*, the NAWC In-House Core Science and Technology Program, CS&T Project *Multichannel Segmentation Techniques*, with partial funding from the ONR 6.2 project *ATR Extraction*, N0001497WX20442.

Approved by
C. F. MARKARIAN, *Head*
Research Department
31 August 2002

Under authority of
M. C. BACHMANN
RDML, US Navy
Commander

Released for publication by
K. HIGGINS
Director for Research and Engineering

NAWCWD Technical Publication 8525

Published by . . . . .Technical Information Division
Collation . . . . Cover, 36 leaves
First printing . . . . .30 copies

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE September 2002	3. REPORT TYPE AND DATES COVERED February 1996 to September 1997

4. TITLE AND SUBTITLE Implementing The Region Growing Method, Part 1: The Piecewise Constant Case (U)	5. FUNDING NUMBERS PE 601152N WU12304016 WU A47D47
6. AUTHOR(S) Jorge M. Martin	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Air Warfare Center Weapons Division China Lake, CA 93555-6100	8. PERFORMING ORGANIZATION REPORT NUMBER NAWCWD TP 8525
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Office of Naval Research 800 N. Quincy Street Arlington, VA 22217	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	
12A. DISTRIBUTION/AVAILABILITY STATEMENT A Statement; public release; distribution unlimited.	12B. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

(U) In this report, we illustrate how one can implement the Region Growing method for minimizing the simplified Mumford and Shah functional in dimension 2, for single- and multichannel-data in the piecewise constant setting. For completeness, we include the theory that shows how to obtain the piecewise approximations in general and the form of the General Merging criterion used in the Region Growing method. We describe in detail the Matlab functions used in the implementation and include the listings of the programs for reference. Thus, this report also serves as documentation for the two-dimensional, piecewise constant program in the single and multichannel settings.

14. SUBJECT TERMS Segmentation, Region Growing Method, Implementation, Piecewise Constant, Multichannel, Matlab Functions, Documentation.	15. NUMBER OF PAGES 68
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT U L

CONTENTS

1. INTRODUCTION

In 1985, Mumford and Shah introduced a mathematical model for solving the segmentation problem (Reference 1). Segmentation and boundary detection algorithms are basic tools for extracting global features out of digitized data.

The Mumford and Shah (M&Sh) functional (References 1 and 2) has the form

$$E(u,K) \equiv \alpha \int_{\Omega-K} \|u-g\|^2 \, d\mu + \int_{\Omega-K} \|\nabla u\|^2 \, d\mu + \lambda \cdot \ell(K), \tag{1}$$

where Ω denotes a rectangle in R^d, $d \in \{1,2,3,\cdots\}$ is the dimension of Ω, $g{:}\Omega \to R^c$ denotes the image (g is μ-measurable), $c \in \{1,2,3,\cdots\}$ is the number of channels, and $u : \Omega \to R^c$ is a smooth approximation to g. The rectangle Ω is decomposed into a finite collection of disjoint open sets O_n (n = 1, 2, ..., N) and their boundaries $K = \bigcup_{n=1}^{N} \partial O_n$; so that $\Omega = \left[\bigcup_{n=1}^{N} O_n\right] \cup K$, and K is sufficiently nice to have a length, denoted by $\ell(K)$. The disjoint open sets O_n (n=1, 2, ..., N) and their boundaries $K = \bigcup_{n=1}^{N} \partial O_n$ together with u are called a *segmentation* of (Ω, g).

The goal is to construct two things:

1. A smoothed ideal image $u : \Omega \to R^c$
2. A set of boundaries $K \subset \Omega$

u and K are found by minimizing the functional E.

The first term on the right-hand side (RHS) of Equation 1 ensures that u is a faithful representation of g, the second term ensures that u is as smooth as possible on each open set O_n (the segments), and the last term prevents the boundaries from growing too large. The technique is *multi-scale*. The parameters α and λ are weighting factors that control the quality of the approximation and the coarseness of the segmentation. The technique allows the extraction of features at different levels of detail (scale). The technique is also multichannel; u and g may be vector valued functions. It can be used to segment images of a scene when registered multiple data channels for the same scene are available. These may be data channels from various sensors, hue channels, or preprocessing channels such as wavelet or other transform channels. Thus, the M&Sh model captures all the essential features that must be considered.

The aim of this report is to present an implementation of the region growing (RG) method introduced in References 3 and 4 for minimizing the simplified M&Sh functional

$$E(u,K) \equiv \int_{\Omega-K} \|u-g\|^2 \, d\mu + \lambda \cdot \ell(K) . \tag{2}$$

Their model assumes that $u:\Omega \to R^c$ is a piecewise constant (PC) function defined on $\bigcup_{n=1}^{N} O_n$, constant on each O_n; $u|O_n \equiv A_n$, where A_n is the *average value* of g on O_n. Thus, the second term in Equation 1 vanishes, leading to Equation 2.

The idea behind the RG method for minimizing Equation 2 is to select two adjacent regions O_i, O_j and merge them; that is, the new average A_{ij} of g on $O_i \cup O_j$ is found; the boundary between O_i and O_j, denoted by $\partial(O_i, O_j)$, is eliminated; and the functional E is evaluated. If E decreases, then O_i and O_j are permanently merged, leading to a coarser segmentation. Otherwise, another pair is considered. Starting with the trivial segmentation, i.e., one where every region consists of only one point (pixel), this procedure arrives at a segmentation with minimal (local) energy for the chosen scale parameter λ. If λ is large, the emphasis is on the length of the boundaries K and the minimization of E leads to fewer boundaries and hence to a coarser segmentation. Conversely, a small λ puts the emphasis on the first term of Equation 2, leading to an approximation u that follows g much more closely and allows for more boundaries. Thus, a small λ leads to a finer segmentation. In the limit when λ is zero, the minimization of E forces $u = g$, leading to the finest possible segmentation, the trivial segmentation where every region consists of a single point (pixel).

The *merging criterion* for merging two adjacent regions O_i and O_j is defined to be the difference between $E(u,K)$ and the new value $E(\tilde{u}, K - \partial(O_i, O_j))$ after a merger.

If $u = A_n$ on each O_n and $\tilde{u} \equiv \begin{cases} A_{ij} \; on \; O_i \cup O_j \\ u \;\; otherwise \end{cases}$, where A_{ij} is the average value of g on $O_i \cup O_j$, then the merging criterion for the PC case (References 3 and 5) is given by

$$\mathrm{M}_{ij} \equiv E(u,K) - E(\tilde{u}, K - \partial(O_i, O_j)) = \lambda \cdot \ell(\partial(O_i, O_j)) - \frac{\mu(O_i)\mu(O_j)}{\mu(O_i) + \mu(O_j)} \cdot \left\| A_i - A_j \right\|^2, \quad (3)$$

where $\mu(O_i)$ denotes the area of the open set O_i. If $\mathrm{M}_{ij} > 0$ there is a decrease in E and the regions O_i, O_j are merged.

While the expression on the RHS of Equation 3 for M_{ij} appears in Reference 3, the authors do not provide a derivation of it. A derivation of it cannot be found in Reference 4 either. Reference 5 presents a complete derivation of Equation 3 and its generalization to the case where the approximations u_n are not PC; that is, the approximations u_n are restrictions to O_n of functions belonging to a class of functions defined as the linear span of a finite set of judiciously chosen functions $f_1, f_2, \ldots, f_\ell$ mapping Ω into R^c. For example, this can include PC, piecewise affine (PA), piecewise polynomial, piecewise exponential, piecewise sinusoidal approximations.

In the general setting of Reference 5, each $u_n = u|O_n$ has the form $u_n = \sum_{k=1}^{\ell} a_{nk} f_k$. If we let $A_n = [a_{n1} \; a_{n2} \cdots a_{n\ell}]^T \in R^\ell$ and $F \equiv [f_1 \vdots f_2 \vdots \cdots \vdots f_\ell]$, then u_n can be written as

$u_n = FA_n$. The weighting factor $\dfrac{\mu(O_i)\mu(O_j)}{\mu(O_i)+\mu(O_j)}$ appearing in Equation 3 becomes a weighting matrix H_{ij}, defined by

$$H_{ij} \equiv M_j[M_i + M_j]^{-1} M_i, \tag{4}$$

where

$$M_n \equiv \int_{O_n} F^T F \, d\mu \qquad (n=1,2,\ldots,N). \tag{5}$$

The merging criterion in this general setting has the form

$$\mathrm{M}_{ij} \equiv E(u,K) - E(\tilde{u}, K - \partial(O_i,O_j)) = \lambda \cdot \ell(\partial(O_i,O_j)) - \left\|A_i - A_j\right\|_{H_{ij}}^2 , \tag{6}$$

where the weighted norm $\|\cdot\|_H$ is defined as $\|x\|_H^2 \equiv x^T H x$ for $x \in R^\ell$.

Reference 5 also includes some examples showing the form that the functions $u_n = FA_n$ and the matrix $F \equiv [f_1 \vdots f_2 \vdots \cdots \vdots f_\ell]$ take in various settings. These include the multichannel PC, the multichannel PA, and the multichannel piecewise quadratic (PQ) settings in various dimensions.

Here we use the results of Reference 5 and illustrate how one can implement the RG method in dimension 2 for PC approximations u_n in the single- and multichannel settings. The multichannel PA implementation is described in Reference 6. Sufficient information can be found in Reference 6 to implement the multichannel PQ algorithm following similar steps.

In Section 2 we state the results of Reference 5 that will be needed here. A tiling problem in the plane that arose as a result of trying to speed up our implementation is discussed in Section 3.

A general description of the program structure is given in Section 4. This section is generic in the sense that all of the different versions of the RG method that we have implemented have the same structure. We have implemented the RG method in the PC single- and multichannel settings in dimension 2 and in the PA single- and multichannel settings in dimensions 2 and 3.

In Section 5 we describe in detail the Matlab functions that implement the single-channel PC-RG method in two dimensions and we indicate the changes involved in the multichannel implementation. These changes are minimal. Listings of the functions are included in the Appendix for reference and for completeness. The changes involved in the multichannel implementation are also listed. Thus, this report serves as documentation for the two dimensional PC-RG programs in the single- and multichannel settings.

2. PIECEWISE APPROXIMATIONS

In this section we show how to obtain the piecewise approximations u on the sets O_n and how to compute the approximation u_{ij} on a union $O_i \cup O_j$ from the approximations on O_i and O_j.

For each $n = 1, 2, \ldots, N,$ let

$$V_n = \int_{O_n} F^T g \, d\mu \ , \tag{7}$$

where μ is a measure on Ω (e.g., Lebesgue measure). Fix i and j in $\{1, 2, \ldots, N\}$ and define $u_{ij}: \Omega \to R^c$ by $u_{ij} = \sum_{k=1}^{\ell} b_k f_k$. Note that if $B = [b_1 \; b_2 \cdots b_\ell]^T$, then $u_{ij} = FB$.

Define $u: \bigcup_{n=1}^{N} O_n \to R^c$ by $u = u_n$ on O_n for all n.

Define $\tilde{u}:\bigcup_{n=1}^{N} O_n \to R^c$ by $\tilde{u} = \begin{cases} u_{ij} \ on\ O_i \cup O_j \\ u \quad otherwise. \end{cases}$

Proposition 1. (Piecewise approximations to g ; The coefficient vectors A_n and B .)

(a) If $u_n = \sum_{k=1}^{\ell} a_{nk} f_k$ approximates g on O_n in the sense of *least mean square error*; that is, if $\{a_{n1}, a_{n2}, \cdots, a_{n\ell}\}$ minimize $\int_{O_n} \|u_n - g\|^2 d\mu$, then the vector of coefficients A_n satisfies the linear system $M_n A_n = V_n \quad (n = 1, 2, \ldots, N)$.

(b) Similarly, if B minimizes $\int_{O_i \cup O_j} \|u_{ij} - g\|^2 d\mu$, then B satisfies the linear system $(M_i + M_j)B = (V_i + V_j)$.

(c) $B = (M_i + M_j)^{-1}(M_i A_i + M_j A_j)$ provided the inverse exists.

Proof. (See Reference 5.)

Proposition 2. (The merging criterion.)

If u and $\tilde{u}$ are as above and $E(u, K) \equiv \int_{\Omega - K} \|u - g\|^2 d\mu + \lambda \cdot \ell(K)$, then Equation 6 holds.

Proof. (See Reference 5.)

2.1 MULTICHANNEL PIECEWISE CONSTANT APPROXIMATIONS

Suppose u is a PC approximation of g; that is, u is constant on each set O_n and there are c channels. In this case $F = I$, the $c \times c$ identity matrix; because if $u_n = [a_{n1}\ a_{n2} \cdots a_{nc}]^T$, a constant vector with c components, then

$$u_n = a_{n1}\begin{bmatrix}1\\0\\0\\\vdots\\0\end{bmatrix} + a_{n2}\begin{bmatrix}0\\1\\0\\\vdots\\0\end{bmatrix} + \cdots + a_{nc}\begin{bmatrix}0\\0\\0\\\vdots\\1\end{bmatrix} = a_{n1}f_1 + a_{n2}f_2 + \cdots + a_{nc}f_c \ .$$

Therefore, $f_k = \begin{bmatrix}0\\\vdots\\1\\\vdots\\0\end{bmatrix} \leftarrow (k^{th} - position)$, $F = [f_1 \vdots f_2 \vdots \cdots \vdots f_c] = \begin{bmatrix}1 & 0 & \cdots & 0\\0 & 1 & \cdots & 0\\\vdots & & \ddots & \\0 & 0 & \cdots & 1\end{bmatrix}$, and $F^T F = I$.

Since $M_n = \int_{O_n} I\, d\mu = \mu(O_n)\cdot I$ for all n, the weight matrix H_{ij} is a multiple of the identity matrix:

$$H_{ij} = \frac{\mu(O_i)\cdot\mu(O_j)}{\mu(O_i)+\mu(O_j)}\cdot I \quad \text{and} \quad \|x\|^2_{H_{ij}} = \frac{\mu(O_i)\cdot\mu(O_j)}{\mu(O_i)+\mu(O_j)}\cdot\|x\|^2 .$$

The merging criterion for the multichannel PC case becomes

$$\lambda\cdot\ell(\partial(O_i,O_j) - \frac{\mu(O_i)\cdot\mu(O_j)}{\mu(O_i)+\mu(O_j)}\cdot\|A_i - A_j\|^2 ,$$

where A_i (A_j respectively) is the vector of values of u on O_i (O_j respectively). Thus, Equation 3 is a special case of Equation 6.

By Proposition 1, A_n satisfies $\mu(O_n)\cdot A_n = \int_{O_n} g\, d\mu$. Thus, $A_n = \frac{1}{\mu(O_n)}\cdot\int_{O_n} g\, d\mu$ is the average value of g on O_n. Similarly, $B = \frac{1}{\mu(O_i)+\mu(O_j)}\cdot\int_{O_i\cup O_j} g\, d\mu$ is the average value of g on $O_i \cup O_j$. In this case, part (c) of Proposition 1 simply shows how to obtain the average of g on $O_i \cup O_j$ from the average of g on O_i and O_j.

As an example, we include here the single channel PA case in dimension 2 for comparison and to emphasize that Equation 6 is indeed necessary beyond the PC setting and that Equation 3 applies only in the PC setting.

2.2. SINGLE-CHANNEL PIECEWISE AFFINE APPROXIMATIONS ($d = 2$)

If $d = 2$, then Ω is a rectangle in R^2. Suppose $g{:}\Omega \to R$ and u is a PA approximation to g. Then

$$u_n(x,y) = a_{n1} \cdot 1 + a_{n2} \cdot x + a_{n3} \cdot y = [1 \;\; x \;\; y] \begin{bmatrix} a_{n1} \\ a_{n2} \\ a_{n3} \end{bmatrix} = F(x,y) A_n .$$

Therefore, $f_1(x,y) \equiv 1$, a constant function, $f_2(x,y) = x$, and $f_3(x,y) = y$, $(x,y) \in \Omega$. The matrix $F(x,y) = [\,1 \quad x \quad y\,]$ is a 1 x 3 matrix, and

$$F^T(x,y)F(x,y) = \begin{bmatrix} 1 \\ x \\ y \end{bmatrix} [1 \;\; x \;\; y] = \begin{bmatrix} 1 & x & y \\ x & x^2 & xy \\ y & yx & y^2 \end{bmatrix}, \qquad (x,y) \in \Omega .$$

$$M_n = \int_{O_n} F^T(x,y)F(x,y)\,dxdy = \int_{O_n} \begin{bmatrix} 1 & x & y \\ x & x^2 & xy \\ y & yx & y^2 \end{bmatrix} dxdy$$

$$= \begin{bmatrix} \int\limits_{O_n} 1\, dxdy & \int\limits_{O_n} x\, dxdy & \int\limits_{O_n} y\, dxdy \\ \int\limits_{O_n} x\, dxdy & \int\limits_{O_n} x^2\, dxdy & \int\limits_{O_n} xy\, dxdy \\ \int\limits_{O_n} y\, dxdy & \int\limits_{O_n} xy\, dxdy & \int\limits_{O_n} y^2\, dxdy \end{bmatrix},$$

so M_n is no longer a multiple of the identity matrix.

$$V_n = \int\limits_{O_n} F^T(x,y) g(x,y)\, dxdy = \begin{bmatrix} \int\limits_{O_n} 1 \cdot g(x,y)\, dxdy \\ \int\limits_{O_n} x \cdot g(x,y)\, dxdy \\ \int\limits_{O_n} y \cdot g(x,y)\, dxdy \end{bmatrix}.$$

The coefficient vector $A_n \equiv \begin{bmatrix} a_{n1} \\ a_{n2} \\ a_{n3} \end{bmatrix}$ satisfies the linear system $M_n A_n = V_n$.

Since the matrices M_n and H_{ij} are no longer multiples of the identity matrix, Equation 3 no longer applies and one must use the general merging criterion in Equation 6. (See Reference 6 for more details on the multichannel PA setting.)

3. TILING PROBLEM

When implementing the RG method, the question of which pair of adjacent regions O_i, O_j should be considered first arises. What procedure should be used to select the sequence of pairs of adjacent regions? In our initial implementation we decided to merge a pair of regions O_i, O_j with maximal criterion M_{ij}. This procedure leads to a steepest-descent type of procedure because the maximal criterion M_{ij} represents the largest decrease in E when two regions are merged. We discovered, however, that this approach often leads to regions that grow too large and with too many small neighboring regions. When regions become such that they have too many neighbors, the procedure slows down considerably. We concluded that, at least initially, one should avoid large regions with too many small neighbors. We opted to use a seeding procedure at the initial stages of the two-dimensional merging process. The seeding procedure (tiling) is described next.

The RG method starts with an initial segmentation of Ω. In the discrete PC case, the initial segmentation can be chosen to be the trivial segmentation, that is, one in which each O_n consists of one single *pixel* (PC case only). Then each pixel has four neighbors: up, down, right, and left neighbor. Note that the diagonal neighbors of a pixel x share a boundary with x (a corner) of zero length. Consequently, these neighbors will never merge with x, because the merging criterion in this case is never positive (see Equation 6).

When a pixel x merges with its four neighbors, it forms a cross-like region (Figure 1) which we will call a *first level tile* (FL-tile). The pixel x is the center of the tile.

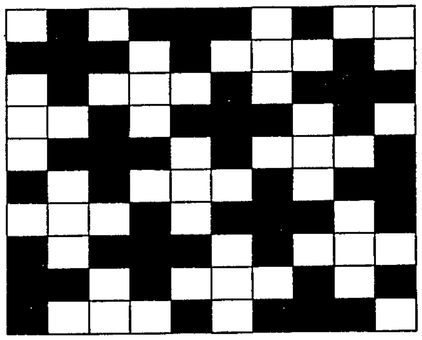

FIGURE 1. First-Level Tiles.

3.1 ITERATED TILING PROBLEM IN TWO DIMENSIONS

The tiling problem for $\Omega \subseteq R^2$ can be stated as follows. (Note that Ω may be replaced by R^2.)

1. Determine the centers of a collection of FL-tiles so that the collection of FL-tiles cover Ω and do not overlap (i.e., intersect only at the boundary). Such a covering will be called a *first-level tiling* of Ω.

2. Determine the centers of a collection of the resulting FL-tiles so that the *second-level tiles* (SL-tiles), that is, a FL-tile merged with its adjacent FL-tile neighbors, cover Ω and do not overlap. Such a covering will be called a *second-level tiling* of Ω.

3. Determine how to iterate the tiling procedure implied by 1 and 2 to obtain higher-level tilings.

3.2 ITERATED TILING PROBLEM SOLUTION IN TWO DIMENSIONS

1. First-level tiling (discrete case).

Let $\Omega = \{$ (i,j)-pixels : $0 \le i \le r$, and $0 \le j \le s$ $\}$. Let

$$a_i = \begin{cases} 3i \pmod 5 & \textit{for } i = 0,1,2,3,4 \\ a_{i-5} & \textit{for } i \ge 5 \end{cases}$$

then, the centers for the FL-tiles can be chosen to be

$$C_1 = \{ c_{ik} = (i, a_i + 5k) : i \in \mathrm{Z}, k \in \mathrm{Z} \} \cap \Omega .$$

Note that the centers in C_1 are distributed with density 1/5 per unit area and the area of a FL-tile is 5. Trivially one can check that FL-tiles with centers in C_1 cover Ω (except for edges) and do not overlap.

2. Second-level tiling.

The centers C_2 for the second level tiling must be chosen from the set C_1. There are several choices. They must be spaced a distance five in both directions. For example:

$$C_2 = \{ c_{ik} = (5i, 5k) : i \in \mathrm{Z}, k \in \mathrm{Z} \} \cap \Omega .$$

Remark 3.1. The process iterates to higher-level tiles. The centers in C_2 form a square lattice just as the centers of the initial pixels do. The second-level tiles are square like (Figure 2). The third level tiles will form a cross-like region (Figure 3). Thus, the process clearly iterates.

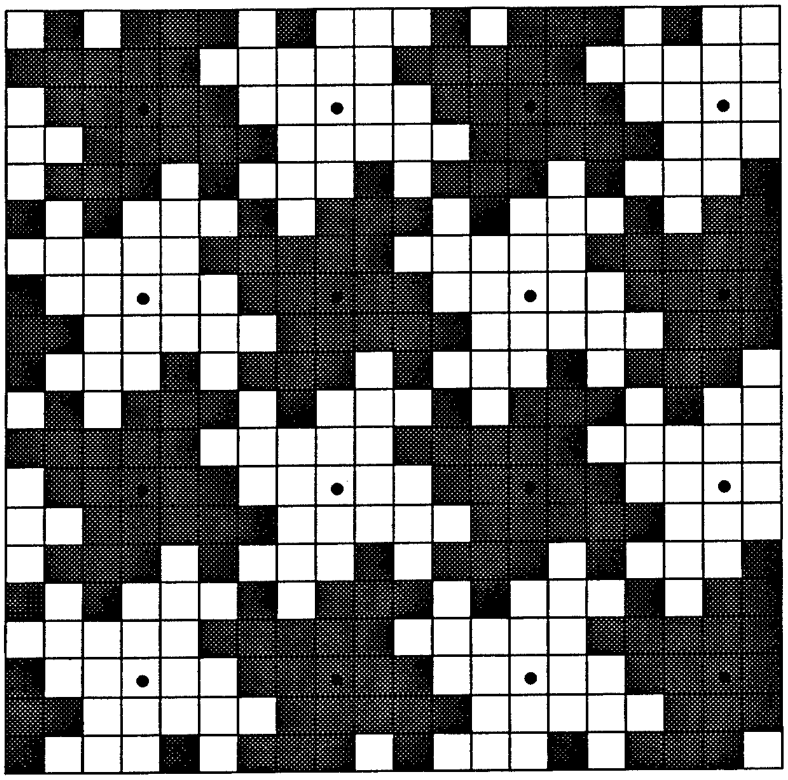

FIGURE 2. Second-Level Tiles.

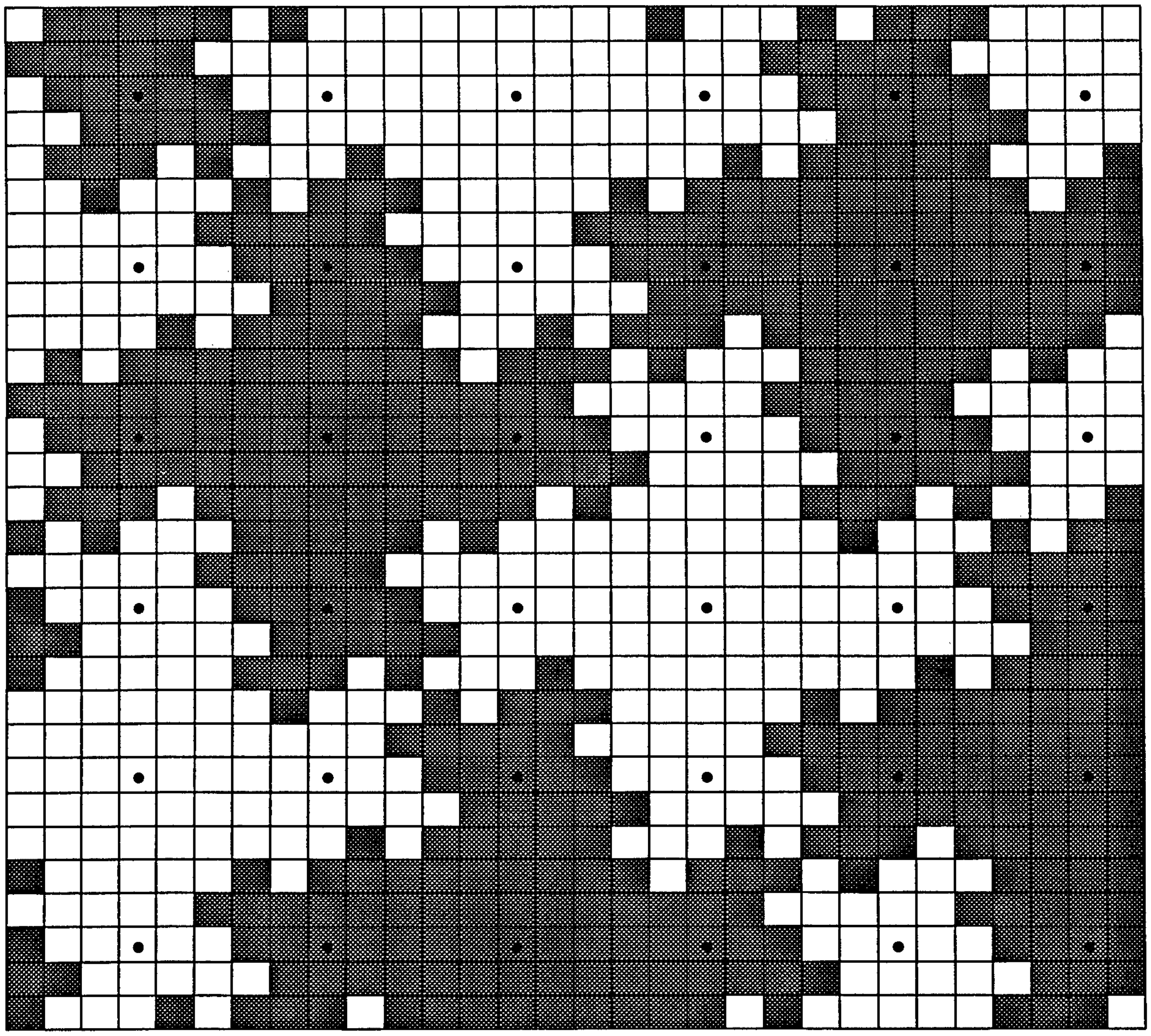

FIGURE 3. Third Level Tiles.

Remark 3.2. The goal of the tiling problem is to prevent regions from having too many small neighbors. In the tiling process previously described, the regions maintain four, and only four, neighbors throughout the process. Furthermore, the area of FL-tiles is 5, the SL-tiles have area 25, and the k-level tiles have area 5^k. Thus, the tile area grows exponentially while the number of neighbors remains constant. This is the case for the worst-case images, which ironically for the RG method are the blank images or homogeneous images (images with constant g) that force the method to merge all the initial regions into one. For non-blank images the iterated tiling process will be corrupted when lower-level tiles do not get merged into higher-level tiles because they are not supposed to (merging criterion ≤ 0). After a few iterations of the tiling process our algorithm starts merging regions with maximal merging criterion (steepest descent) to arrive at a minimum of E.

4. PROGRAM STRUCTURE GENERAL DESCRIPTION

The RG method arrives at a segmented image by merging adjacent regions with a positive merging criterion (Equation 6) until a minimum of the simplified M&Sh functional (Equation 2) is reached. To evaluate Equation 6, we need the coefficient vectors A_i and A_j, the two matrices M_i and M_j, and the length of the common boundary $\ell(\partial(O_i, O_j))$. After merging two regions O_i and O_j, we also need the two vectors V_i and V_j in order to compute the new coefficient vector $A_{ij} = (M_i + M_j)^{-1}(V_i + V_j)$ for the approximation to g on $O_i \cup O_j$.

All the versions of the RG algorithm that we have implemented keep track of several lists of objects that are associated with each region. Each region k is associated with five lists and three objects. The three objects are the coefficient vector A_k, the vector V_k, and the matrix M_k associated with region k.

The five lists associated with region k are

1. A list of neighbors for region k

2. A list with the lengths of the common boundaries between region *k* and each of its neighbors
3. A list with the values of the merging criterion between region *k* and each of its neighbors
4. A list with the boundary pixel labels between region *k* and each of its neighbors
5. A list of the pixels in region *k*

These lists and objects must be updated every time two regions merge. Updating lists 1, 4, and 5 involves taking the union of two sets. The pixels in the new region formed are simply the union of the pixels in the two merged regions. This union is implemented by the function **merge_pix** and is discussed in more detail in subsequent paragraphs. The list of boundary pixels of the new region is the union of the boundary pixels of the two merged regions after deleting, from both lists, the pixels belonging to the common boundary. This operation is implemented by the function **New_Boundary**. When two regions merge, say region A and region B, the list of neighbors for the new region is the union of the two lists of neighbors (list A and list B) after deleting the following:

1. B from list A
2. A from list B
3. All common neighbors from list B

The operations 1, 2, and 3 for updating the list of neighbors for the new region are implemented by the function **mrg_N_B** and the functions that **mrg_N_B** calls. The new region keeps the label A and the new list is list A. Region B is said to have been absorbed by region A.

Every neighbor of a region B lists B as one of its neighbors. When a region B gets absorbed by region A, B no longer exists and must be deleted from the lists of every one of its neighbors and substituted by A, unless the neighbor is a neighbor of both A and B, in which case we delete only B from its list. This operation is performed by the function **replace_B_by_A**. This function also updates the merging criterion because the area of the neighbor A now includes the area of B as well. As the different lists of neighbors are updated, so are the merging criteria. The common neighbors are dealt with by **mrg_N_B**.

The length of the boundary between two regions changes only when the two regions A and B being merged have a common neighbor C. In this case, if A absorbs B, then the length of the boundary between the new A and C is simply the sum of the lengths of boundary between C and the old A and between C and B. Updating the boundary lengths is also taken care of by the function **mrg_N_B**.

The coefficient vectors A and V, the matrix M and the preceding lists 1 through 4 are also updated in **mrg_N_B** and the functions that **mrg_N_B** calls.

Figure 4 shows the structure of the PC-RG program for two-dimensional imagery. The program, called **Tile_Max_Reg_Grow_B**, is divided into four parts.

In Part I, the five lists and the three objects associated with each initial region are defined for every region in the initial segmentation. In the PC case the initial segmentation is chosen to be the trivial segmentation where each region consists of a single pixel. Thus, for an nxm-image, there are nm initial regions in the PC case. Each region is labeled with an integer k; $1 \leq k \leq nm$. The function **initialize** defines the initial pixels in each region. The function **NGB_PNTR_init_B** defines the initial lists of neighbors, common boundary lengths, merging criteria, boundary pixels, and the initial A_k, V_k, and M_k for each region $k \in \{1,2,\cdots,nm\}$.

In Part II of **Tile_Max_Reg_Grow_B**, the centers of the first four levels of tiles are defined by the function **tile** and the two functions that it calls. The function **sweep_reg_Tile_B** sweeps through the centers of the tiles in order to merge them with their neighbors according to the merging criterion up to the highest level of tiles chosen.

Part III finishes the merging of regions by selecting pairs of regions with maximal merging criterion (steepest descent) for merging until all the remaining criteria lie below the chosen non-negative threshold. The resulting sets of regions, boundaries, and coefficients constitute the segmentation of the original image.

In Part IV, the piecewise approximation to the original image is computed and the information is assembled properly for display. It is possible to display, for example, individual regions or individual boundaries if desired, as well as the complete set of boundaries and the complete piecewise approximation to the image.

We note here that Part II can be skipped if one prefers a pure steepest-descent procedure for minimizing Equation 2. This may lead to a different local minimum for Equation 2. Part II was introduced to speed up the algorithm.

This finishes the general description of the structure of our programs. Next, we go into a detailed description of each function involved.

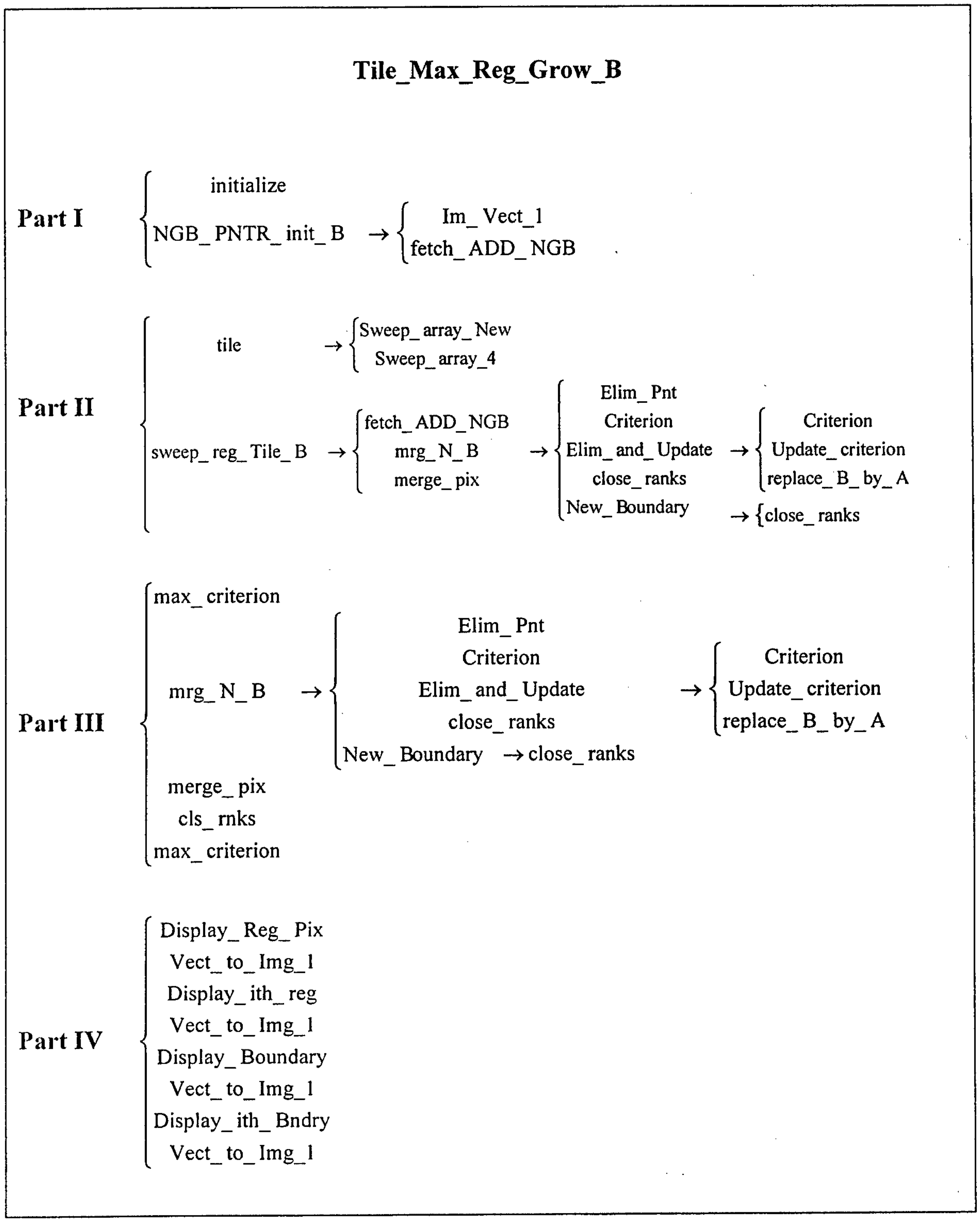

FIGURE 4. The Piecewise Constant Matlab Program.

5. MATLAB IMPLEMENTATION OF PC-RG METHOD

5.1 PART I. INITIALIZING

In Part I we describe the functions **initialize, NGB_PNTR_init_B, Im_Vect_1**, and **fetch_ADD_NGB**.

$$\begin{cases} \text{initialize} \\ \text{NGB\_PNTR\_init\_B} \rightarrow \begin{cases} \text{Im\_Vect\_1} \\ \text{fetch\_ADD\_NGB} \end{cases} \end{cases}$$

5.1.1 Function *initialize*

The input to this function is *nm*. The outputs are the six arrays: *Start*, *End*, *Reg_Pix*, *P_S*, *Strt_S*, and *End_S*.

This simple function initializes six arrays. To understand the role of these arrays, we should explain here how we have implemented the operation of union of two disjoint sets in Matlab.

Suppose we have four disjoint sets labeled A, B, C, D, and suppose the labels are positive integers, say $A = 1$, $B = 2$, $C = 3$, and $D = 4$. Suppose the elements of each set are numbers also, say $A = \{a_1, a_2, a_3\}$, $B = \{b_1, b_2, b_3\}$, $C = \{c_1, c_2, c_3, c_4\}$, and $D = \{d_1, d_2, d_3, d_4\}$. To implement the possible unions of these four sets (e.g. $A \cup C$ or $B \cup D$) we use four arrays. The first array contains the elements of all four sets; call it *Elements*:

$$Elements = [a_1, a_2, a_3, b_1, b_2, b_3, c_1, c_2, c_3, c_4, d_1, d_2, d_3, d_4].$$

The second array has four elements (the number of sets) and indicates where, in the array *Elements*, the members of each set start; call it *Start*:

$$Start = [\ 1, 4, 7, 11\ ].$$

So, the elements of set A start at $Start(A) = Start(1) = 1$, the elements of set B start at $Start(B) = Start(2) = 4$, etc. Thus, the elements of *Start* are pointers to the beginning of the list of elements of each set contained in the array *Elements*.

The third array indicates where, in *Elements*, the members of each set end; call it *End*:

$$End = [\ 3, 6, 10, 14\ ].$$

So, the elements of the set $C = 3$ start at $Start(C) = 7$ and end at $End(C) = 10$.

The fourth array is a pointer to the next element; call it *Pointer*. In this example, we have

$$Pointer = [\ 2, 3, 0, 5, 6, 0, 8, 9, 10, 0, 12, 13, 14, 0\ ].$$

The elements of a set can be retrieved from the array *Elements* using the two arrays *Start* and *Pointer*. For example, the elements of set $B = 2$ are

$$Elements(Start(2)) = Elements(4) = b_1$$
$$Elements(Pointer(Start(2))) = Elements(Pointer(4)) = Elements(5) = b_2$$
$$Elements(Pointer(Pointer(Start(2)))) = Elements(Pointer(5)) = b_3$$

Note that $Pointer(6) = 0$ and the zero indicates the end of the list for set B. Thus, the idea is to iterate along the array *Pointer*, starting at $Start(B)$, until the value zero is reached: $Start(B)$, $Pointer(Start(B))$, $Pointer(Pointer(Start(B)))$, ... , 0. Also note that for any set X, $Start(X)$ is not only the index of *Elements* corresponding to the first element of X, but is also the index for starting the iteration along the array *Pointer* to retrieve the elements of the set X.

In general, the arrays *Start*, *End*, and *Pointer* are defined so that *Start*(X) is the index of the first element of X, *Pointer*(*Start*(X)) is the index of the second element of X, *Pointer*(*Pointer*(*Start*(X))) is the third, and so on,

$$\mathit{Pointer}(\mathit{End}(X)) = 0 \quad \text{for all sets } X \tag{5.1}$$

and *End*(X) is the index of the last element of X. When a set X disappears, we set *Start*(X) = 0 to indicate that it no longer plays a role. One can go through the elements of a set or a list of elements using the following Matlab while loop.

```
                      % The elements of set C
x=Start(C);           % Pointer to the first element in the list C or set C
while x ~= 0          % If Start(C) is not zero, the set C is still viable
                      % If x is not zero, there is another element in the set C
    Elements(x)       % An element of set C
    x=Pointer(x);     % Advance the pointer to the index of the next element
end                   % When x = 0 the list has ended
```

The union of two sets, that is, an assignment of the form $A = A \cup C$, for example, can be implemented by concatenating the list for set A with the list for set C, as follows.

1. $\mathit{Pointer}(\mathit{End}(A)) = \mathit{Start}(C)$
2. $\mathit{End}(A) = \mathit{End}(C)$
3. $\mathit{Start}(C) = 0$

Statement 1 concatenates the two lists. Before Statement 1, $\mathit{Pointer}(\mathit{End}(A)) = 0$ indicates the end of list A. By setting $\mathit{Pointer}(\mathit{End}(A)) = \mathit{Start}(C)$, the list continues through the elements in the list C. Statement 2 moves the end of list A to the end of list C. $\mathit{Start}(C) = 0$ in Statement 3, is for bookkeeping purposes; it indicates that the set C has been merged with another set and no longer exists.

In the two-dimensional setting, the image is an array of dimensions $n \times m$. The integers n and m denote the dimensions of the image throughout the paper and the programs.

As mentioned in the preceding paragraphs, the function **initialize** initializes six arrays. The first three arrays (*Reg_Pix*, *Start*, and *End*) are used to keep track of the pixels belonging to each region and to implement the unions of regions as the merging of regions proceeds. Since the image is $n \times m$, there are *nm* pixels. The pixels are labeled 1 through *nm* (Figure 5), starting with the first row of pixels. Pixel (*i*, *j*) receives the label $k = (i-1)m + j$.

$$\begin{bmatrix} 1 & 2 & 3 & \cdots & m \\ m+1 & m+2 & m+3 & \cdots & 2m \\ 2m+1 & 2m+2 & 2m+3 & \cdots & 3m \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ (n-1)m+1 & (n-1)m+2 & (n-1)m+3 & \cdots & nm \end{bmatrix}$$

FIGURE 5. Pixel Labels.

In the PC case, the initial segmentation is the trivial segmentation. Each region consists of one pixel; hence, there are *nm* regions. The regions are labeled 1 through *nm* as the pixels, so that *Region(i)* = { *i* }, *i* = 1, 2, . . . , *nm*.

The arrays *Start* and *End* in **initialize**, as discussed previously, indicate where the pixels of each region start and end. Since each region has one element,

Start = *End* = [1, 2, 3, . . . , *nm*].

The array *Reg_Pix*, the pointer associated with the region labels and pixel labels, is a 1 x *nm* array and, according to Equation 5.1 and the definition of *End*, *Reg_Pix* = [0, 0, . . . , 0]. Moreover, the array *Reg_Pix* also plays the role of the array *Elements*. In this setting the array *Elements* is [1, 2, 3, . . . , *nm*], so it is not necessary to explicitly define it. To keep track of the pixels in each region, we need only *Start*, *End*, and *Reg_Pix*.

The other three arrays defined in the function **initialize** are called *P_S*, *Strt_S*, and *End_S*. These are used to keep track of the non-zero entries in the array *Start.* A zero entry in *Start* indicates that the region corresponding to that entry has been absorbed by (merged with) another region and will no longer be considered. Thus, by keeping track of the non-zero entries in *Start* one can search efficiently through the remaining regions. The array *P_S* is the pointer, *Strt_S* and *End_S* are the start- and end-arrays associated with the pointer *P_S*. Thus, initially *Strt_S* = 1, *End_S* = *nm*, and *P_S* = [2, 3, 4, . . . , *nm*, 0]. The function **cls_rnks** updates these three arrays so that *P_S* will skip over the zero entries in *Start*.

5.1.2 Function *NGB_PNTR_init_B*

The function **NGB_PNTR_init_B,** one of the main functions in our implementation of the RG method, initializes 11 arrays:

1. The array *M*, a 1 x (*nm*) array
2. The array *V*, a ***c*** x (*nm*) array
3. The array *C*, a ***c*** x(*nm*) array, where ***c*** is the number of channels
4. The array *Reg_N*, a 3 x (4*nm*) array
5. The array *Start_N*, a 1 x (*nm*) array
6. The array *End_N*, a 1 x (*nm*) array
7. The array *Ptr_N*, a 1 x (4*nm*) array
8. The array *Reg_B*, a 1 x (4*nm*) array
9. The array *Start_B*, a 1 x (*nm*) array
10. The array *End_B*, a 1 x (*nm*) array
11. The array *Prt_B*, a 1 x (4*nm*) array

With each region *k*, a function $u_k = \sum_{i=1}^{\ell} a_{ki} f_i$ approximates the image g on O_k. The coefficient vector $A_k = [a_{k1}\ a_{k2} \cdots a_{k\ell}]^T \in R^{\ell}$ satisfies $M_k A_k = V_k$ (see *Proposition 1*).

In the PC setting, the matrix $M_k = \mu(O_k) \cdot I$, where $\mu(O_k)$ denotes the area of region O_k. The 1x(nm) array M contains the areas of all the initial regions which are all equal to one.

In the discrete case, when O_k is one pixel, say pixel k, the vector $V_k = \int_{O_k} F^T g \, d\mu$ becomes $V_k = \sum_{i \in O_k} F^T(i) g(i) = \sum_{i \in O_k} g(i) = g(k)$ and since $M_k = I$, $A_k = V_k = g(k)$, for all k. Here, the c x nm array C contains all the initial coefficients $A_k = g(k)$ $(1 \le k \le nm)$, where c is the number of channels. Thus, $C = V$ and V is a c x nm array with $V(:,k) = g(\text{pixel } k)$.

The function **Img_Vect_1** transforms a 1-channel, *(n* x *m)*-image into a (1 x *nm)* array *V*. For multichannel data we use **Img_Vect_1** on each channel to transform a *c*-channel, *(n* x *m)*-image into a *(c* x *nm)* array *V* as follows.

For multichannel data use

```
for i=1:ch
    V(i,:)=Img_Vect_1(n,m,CHANNEL(i));
end
```

The arrays *Start_N*, *End_N*, and *Ptr_N* are, respectively, the start, end, and pointer arrays associated with the 3 x (4*nm*) array *Reg_N*, which contains the neighbors, boundary lengths, and merging criteria for each region *k*. This array plays the role of the array *Elements* discussed as an example in the description of the function **initialize.**

Every pixel *k* (except the pixels at the edges of the image) has four neighboring pixels: an up, down, left, and right neighbor (see Section 3). Each neighbor has a label in the set $\{1,2,3,\cdots,nm\}$. The 3 x 4*nm* array *Reg_N* contains, in the first row, the labels of the neighbors of each pixel. The label 0 is given to neighbors that do not exist. The neighbors are labeled counterclockwise starting with the upper neighbor: up, left, down, and right. So, for example (see Figure 5), the neighbors of pixel 1 are 0, 0, *m*+1, 2. The neighbors of pixel *m*+2 are 2, *m*+1, 2*m*+2, *m*+3. The neighbors are listed, four per pixel, along the first row of *Reg_N* in the same order as the pixel labels.

On the second row of *Reg_N* and in the same order are listed the lengths of the common boundaries between a region and its four neighbors multiplied by lambda (see Equation 3). Since all the regions consist of one pixel initially, all the lengths equal one. Along the third row of *Reg_N*, and in the same order as the neighbors, are listed the values of the merging criteria between a region *k* and its four neighbors. The initial merging criteria in the PC-setting are

$$lambda - \frac{1}{2}[C(k) - C(neighbor)]^T [C(k) - C(neighbor)], \qquad (5.2)$$

where *C(k)* are column vectors of coefficients (corresponding to the coefficient vectors A_k of Equations 3 and 6) of dimension ***c*** (number of channels) associated with region *k*. Since the Expression 5.2 involves an inner product of ***c***-dimension vectors, the Matlab statement that implements it is already multichannel. However, one must use *C(:,k)* and *C(:,neighbor)*. Here, we call the function **fetch_ADD_NGB** to fetch the neighbors of each region *k*. It is interesting that the function **fetch_ADD_NGB** uses *Start_N*, *Ptr_N*, and *Reg_N* as inputs, and these arrays are being defined here in the calling function. It works. The parts that **fetch_ADD_NGB** needs are defined before it is called.

Since each region has four neighbors, the list of neighbors for region *k* ends at *Reg_N(1, 4k)* and starts at *Reg_N(1, 4k-3)* $(1 \le k \le nm)$. Thus, *End_N(k)* = *4k* and *Start_N(k) = 4k-3* $(1 \le k \le nm)$. The pointer *Ptr_N* is defined as

$$Ptr\_N = [\,2, 3, 4, 0, 6, 7, 8, 0, 10, 11, 12, 0, \ldots, 4nm\text{ -}2, 4nm\text{ -}1, 4nm\,, 0].$$

The arrays *Start_B*, *End_B*, and *Ptr_B* are, respectively, the start, end, and pointer arrays associated with the 1 x (4*nm*) array *Reg_B*, which contains the labels of the boundaries associated with each region *k*. This array plays the role of the array *Elements* discussed as an example in the description of the function **initialize.**

Between any two adjacent pixels we consider the existence of a virtual boundary (virtual in the sense that there is no pixel corresponding to the boundary) and divide the boundaries into two classes: vertical and horizontal. The vertical boundaries are labeled 1 through *nm*. The horizontal boundaries are labeled *nm* + 1 through 2*nm*. The number of boundaries is fewer than the labels because the boundaries of the image are not considered boundaries of pixels. Thus, not all the labels are used.

Except for the pixels at the edges of the image, every pixel *k* has four boundaries: an upper boundary labeled *(nm+k-m)*, a left boundary labeled *(k-1)*, a down boundary labeled *(nm+k)*, and a right boundary labeled *k*. The nonexisting boundaries are labeled 0. We note here that one can label the horizontal virtual boundaries 1 through *nm* as well. Different labels are not necessary (see the remark in Section 5.4.4).

Reg_B is a 1x(4*nm*) array with the labels of the four boundaries of each pixel in the same order as the neighbors. Thus, *Start_B* = *Start_N*, *End_B* = *End_N*, and *Ptr_B* = *Ptr_N*.

5.1.3 Function *Img_Vect_1*

This function transforms a single-channel *nxm* image into a (1 x *nm)* vector *V*:

$$V(1,(i-1)m+j) = \text{Image}(i,j)\,, 1 \le i \le n\,,\ 1 \le j \le m.$$

In the multichannel setting, this function must be modified to produce a (*c* x *nm*) vector *V,* where *c* is the number of channels or else the function is used on each channel one at a time.

5.1.4 Function *fetch_ADD_NGB*

This function retrieves the list of addresses of the neighbors of a region *A* and the list of neighbors, boundary lengths, and merging criteria using the arrays *Start_N*, *Ptr_N*, and *Reg_N*. The inputs to the function are *A*, *Start_N*, *Ptr_N*, and *Reg_N*. The function returns four arrays: an array *Addresses* with the addresses of the neighbors of region *A*, an array *Neighbs* with the labels of the non zero neighbors of A, and the arrays *Lengths* and *Criter* with the lengths of boundaries and merging criteria, respectively.

5.2 PART II. TILING

$$\left\{\begin{array}{l} \text{tile} \rightarrow \left\{\begin{array}{l}\text{Sweep\_array\_New}\\ \text{Sweep\_array\_4}\end{array}\right. \\ \text{sweep\_reg\_Tile\_B} \rightarrow \left\{\begin{array}{l}\text{fetch\_ADD\_NGB}\\ \text{mrg\_N\_B}\\ \text{merge\_pix}\end{array}\right. \rightarrow \left\{\begin{array}{l} \text{Elim\_Pnt}\\ \text{Criterion}\\ \text{Elim\_and\_Update} \rightarrow \left\{\begin{array}{l}\text{Criterion}\\ \text{Update\_criterion}\\ \text{replace\_B\_by\_A}\end{array}\right.\\ \text{close\_ranks}\\ \text{New\_Boundary} \rightarrow \{\text{close\_ranks}\end{array}\right.\end{array}\right.$$

Part II of the PC-RG program in two dimensions implements the tiling described in Section 3, which was introduced to speed up the algorithm. Here we describe the functions **tile**, **Sweep_array_New**, **Sweep_array_4**, and **sweep_reg_Tile_B**. The function **fetch_ADD_NGB** was described in Part I, and the rest of the functions are described in Part III.

5.2.1 Functions *tile*, *Sweep_array_New*, and *Sweep_array_4*

The function **tile** simply calls the two functions **Sweep_array_New** and **Sweep_array_4** to create two arrays, *T1* and *T2*. These arrays contain the centers of the levels 1, 2, 3, and 4 tiles (see Section 3). This function has three inputs: the dimensions of the image *n* and *m* and *level* $\in \{1, 2, 3, 4\}$, which is used to select the highest level of tiles desired. If *level* ≤ 2, then only *T1* is created by **Sweep_array_New**. If *level* > 2, then **Sweep_array_4** creates *T2*. The centers of the first two-level tiles are defined in *T1* (the first level in row 1 and the second level in row 2). The centers of the third- and fourth-level tiles are in *T2* (the third level in row 1 and the forth level in row 2). The outputs of this function are *T1* and *T2*.

5.2.2 Function *sweep_reg_Tile_B*

The inputs to this function are *n*, *m*, *M, V, C, T, T1, T2, level, S_N, E_N, P_N, R_N, S, E, RP, S_B, E_B, P_B,* and *R_B*.

This function, which forms the tiles, goes through the centers of the various-level tiles in *T1* and *T2* according to the level of tiling selected (input *level*) and merges the center of a tile with its adjacent neighbors (to form the tile), if the merging criterion exceeds the chosen threshold *T*. The steps are as follows.

A center is selected and labeled *A*:

$A = T(1,k)$ for level 1, $k = 1, 2, 3, \ldots$, or
$A = T(2,k)$ for level 2, $k = 1, 2, 3, \ldots$, or
$A = T(3,k)$ for level 3, $k = 1, 2, 3, \ldots$, or
$A = T(4,k)$ for level 4, $k = 1, 2, 3, \ldots$.

If $A \neq 0$, then **sweep_reg_Tile_B** calls the function **fetch_ADD_NGB** (described in Section 5.1) to get the neighbors of *A*. Then it loops through the list of neighbors, checking if the merging criterion exceeds the chosen threshold *T* for merging, and if it does, it calls the function **mrg_N_B** to update the lists and objects associated with *A* and its neighbors. Then it calls the function **merge_pix** to implement the union of the pixels of *A* and the neighbor being absorbed. It returns the updated arrays *M, V, C, S_N, E_N, P_N, R_N, S, E, R_P, S_B, E_B, P_B, R_B.*

The function **mrg_N_B** and the functions it calls and the function **merge_pix** are discussed in Part III.

5.3 PART III. STEEPEST DESCENT

Part III implements the Steepest Descent procedure for minimizing Equation 2.

$$\left\{\begin{array}{l} \text{max\_criterion}\\ \text{mrg\_N\_B}\\ \text{merge\_pix} \rightarrow \left\{\begin{array}{l} \text{Elim\_Pnt}\\ \text{Criterion}\\ \text{Elim\_and\_Update} \rightarrow \left\{\begin{array}{l}\text{Criterion}\\ \text{Update\_criterion}\\ \text{replace\_B\_by\_A}\end{array}\right.\\ \text{close\_ranks}\\ \text{New\_Boundary} \rightarrow \{\text{close\_ranks}\end{array}\right.\\ \text{cls\_rnks}\\ \text{max\_criterion}\end{array}\right.$$

5.3.1 Function *max_criterion*

The inputs to **max_criterion** are *Start_S*, *Ptr_S*, *Start_N*, *Ptr_N*, and *Reg_N*. The outputs are *Region*, *Neighbor*, and *MAX_Criter*.

This function uses a double while loop to search through the remaining regions (using *Start_S* and *Ptr_S* discussed in Section 5.1.1 as *Strt_S* and *P_S*) and through each remaining neighbor of each region (using *Start_N* and *Ptr_N*) in order to find the pair of remaining neighboring regions with maximal criterion. The function returns the selected region-neighbor pair and the value of the maximal criterion.

5.3.2 Function *mrg_N_B*

This function implements most of the operations involved when two regions merge. The inputs are *A*, *B*, *M*, *V*, *C*, *Start_N*, *End_N*, *Ptr_N*, *Reg_N*, *Start_B*, *End_B*, *Ptr_B*, and *Reg_B*. The two regions to be merged are *A* and *B*. The new region formed keeps the label *A*; that is, *A* will absorb *B*. Both *A* and *B* are integers between 1 and *nm*.

The arrays *M*, *V*, and *C* contain, respectively, the matrix M_k, the vector V_k and the vector of coefficients C_k (see *Proposition 1*; we use C_k instead of A_k) associated with every region k $(1 \leq k \leq nm)$. In the 1-channel setting, these three objects are scalars. Thus, *M*, *V*, and *C* are 1 x *nm* vectors. In Part 1 of **mrg_N_B** the values of these scalars are computed for the new region. The new region keeps the label A. The update of these objects in the single-channel case reads as follows:

$M(A) = M(A) + M(B)$, a scalar
$V(A) = V(A) + V(B)$, a scalar
$C(A) = V(A)/M(A)$, a scalar

In the multichannel case (*c* channels), the matrices M_k are simply the (*c* x *c*)-identity matrix multiplied by the area of the region. Therefore, they can be represented by a scalar as in the 1-channel case. The vectors V_k and C_k have dimension *c*. Thus, *V* and *C* are (*c* x *nm*) matrices, where $V(: , k)$ and $C(: , k)$ are column vectors corresponding to region *k*. The update of these objects in the multichannel case reads as follows:

$M(A) = M(A) + M(B)$, a scalar
$V(: , A) = V(: , A) + V(: , B)$, a column vector of dimension *c*
$C(: , A) = C(: , A)/M(A)$, a column vector of dimension *c*

Note that no matrix inversion is required. This modification is the only one needed in **mrg_N_B** to implement the multichannel PC algorithm.

In Parts 2 and 3 of **mrg_N_B**, the array *Reg_N* is updated. As mentioned in Section 5.1.2, *Reg_N* is a 3 x 4*nm* array. The first row of *Reg_N* contains the lists of neighbors of every region; the second row contains the lists of common boundary lengths between every region and its neighbors. The third row contains the lists of merging criteria between every region and its neighbors. When talking about a list of neighbors, lengths or criteria of a given region *A*, we refer to it as list(*A*) for short. It will be clear from the context what kind of list it refers to. The length of the common boundary between two regions, *A* and *B* say, is referred to by length(*A*,*B*). Similarly, the merging criterion between two regions, *A* and *B* say, is referred to by criterion(*A*,*B*).

In Part 2, a double while loop is used to search through list(*A*) and list(*B*) for a common neighbor *CN*. When a common neighbor *CN* is found, it is deleted from list(*B*) to avoid duplication of *CN* in list($A \cup B$) when we merge the two lists. Next, a call to the function **Elim_pnt** (described in the following text) eliminates *B* from list(*CN*), for *B* has been absorbed and no longer exists. Since *CN* is a common neighbor, *A* does not need to be added to list(*CN*); however, since *CN* is a neighbor common to *A* and *B*, length(*A*,*CN*) has increased and it will have to be updated in list(*A*) and list(*CN*). After the merger of *A* and *B*, we have

$$\text{length}(A, CN) = \text{length}(A, CN) + \text{length}(B, CN).$$

The function **Elim_Pnt** also returns a pointer to *A* in list(*CN*) in order to update length(*A*,*CN*) in list(*CN*). Once this pointer is available, length(*A*,*CN*) in list(*CN*) is updated, after updating length(*A*,*CN*) in list(*A*).

Since length(*A*,*CN*) has increased and the area of *A* has increased, the criterion(*A*,*CN*) has changed. A call to the function **Criterion** (described in a subsequent paragraph) updates criterion(*A*,*CN*) in list(*A*). Then, using the pointer to *A* in list(*CN*) previously provided by **Elim_Pnt,** we update criterion(*A*,*CN*) in list(*CN*). This takes care of the common neighbors. We suspect that it is not necessary to update criterion(*A*,*CN*) in list(*A*) and list(*CN*) here, for it will probably be taken care of in Part 3, where we update the criteria between *A* and the rest of its neighbors that were not common to both *A* and *B*.

As mentioned in the preceding text, Part 3 takes care of the list-updating for those neighbors of *A* and *B* that were not common neighbors. Part 3 also updates the lists of the members of list(*B*). This is done by a call to the function **Elim_and_Update** and is explained subsequently in the description of this function.

Once all the lists of have been updated and the lists of neighbors of *A* and *B* are ready to be merged, in Part 4 of **mrg_N_B** the lists of the neighbors of *A* and *B* are concatenated as described in Section 5.1.1. Next, a call to the function **close_ranks** adjusts the pointer *Ptr_N* to skip over the addresses that correspond to regions in list(*A*) that have been absorbed and adjusts *Start_N* and *End_N*. Finally, the boundary of the new region is adjusted with a call to the function **New_Boundary**. These last two functions are also subsequently described.

5.3.2.1 Function *Elim_Pnt*. The inputs to the function **Elim_Pnt** are *C*, *B*, *A*, *Start_N*, *Ptr_N*, and *Reg_N*. Using *Start_N*, *Ptr_N*, *Reg_N*, and the searching technique described in Section 5.1.1, this function searches through the list of neighbors of region *C* looking for regions *B* and *A*. When it finds *B*, it eliminates *B* from the list by setting the entry to zero; and when it finds *A*, it saves the pointer to *A* and returns it to the calling function *(pnt_to_A_inC)*.

5.3.2.2 Function *Criterion*. The function **Criterion** has inputs *A*, *C*, *pointer*, *M*, *Coeff*, and *Reg_N*. The input *pointer* is a pointer to region *C* in the list of neighbors list(*A*) that resides in *Reg_N*. After computing the new merging criterion between regions *A* and *C*, it places it in the list of criteria in list(*A*) using *pointer* and *Reg_N(3,:)*, namely, *Reg_N(3,pointer)*. Since *Coeff*, the input of coefficients for the piecewise approximation can be a column vector, the function **Criterion** is already multichannel. The function returns the updated array *Reg_N*.

5.3.2.3 Function *Elim_and_Update*. Before merging regions *A* and *B*, this function updates list(*A*), the lists of the neighbors of *A*, list(*B*), and the lists of the neighbors of *B* as follows.

Updating list(*A*) and the lists of the neighbors of *A*,

1. Eliminates *B* as neighbor from the list(*A*).

2. Updates criterion(*A*,*C*) for every neighbor *C* in list(*A*), except neighbor *B*, by calling the function **Criterion**.

3. Updates criterion(*A*,*C*) in list(*C*) for every neighbor *C* in list(A), except neighbor *B*, by calling the function **Update_criterion**.

Updating list(B) and the lists of the neighbors of B,

1. Eliminates A as neighbor from the list(B).

2. Updates criterion(A,C) for every neighbor C in list(B), except neighbor A, by calling the function **Criterion**.

3. Replaces B by A in list(C) for all the neighbors C of B in list(B) except neighbor A, and updates criterion(A,C) in list(C) by calling the function **replace_B_by_A**.

Note that if C is a common neighbor of A and B, then B will have been replaced by 0 at the common neighbor stage (i.e., function **Elim_Pnt**). Thus, A will not be duplicated in the list(C).

5.3.2.3.1 Function *Update_criterion*. The inputs to this function are *C*, *A*, *M*, *Coeff*, *Start_N*, *Ptr_N*, and *Reg_N*. Using *Start_N*, *Ptr_N*, *Reg_N*, and the searching technique described in Section 5.1.1, the function searches through the list of neighbors of region C looking for region A. When it finds A (*Reg_N(*1, x) = A), it updates the criterion(C,A) in the array *Reg_N*, namely, *Reg_N(*3,x) = criterion(C,A), by calling the function **Criterion**.

5.3.2.3.2 Function *replace_B_by_A*. The inputs to this function are *A*, *B*, *C*, *Coeff*, *M*, *Start_N*, *Ptr_N*, and *Reg_N*. Using *Start_N*, *Ptr_N*, *Reg_N*, and the searching technique described in Section 5.1.1, the function searches through the list of neighbors of region C looking for region B. When it finds B, it replaces B by A and updates the criterion(C,A) in list(C) contained in the array *Reg_N* by calling the function **Criterion**.

5.3.2.4 Function *close_ranks*. The inputs to function **close_ranks** are *A*, *Start_N*, *End_N*, *Ptr_N*, and *Reg_N*. This function modifies *Start_N(A)*, *End_N(A)*, and the part of the array *Ptr_N* that corresponds to the list of neighbors of A. This is done in order to skip over the addresses in the pointer *Ptr_N* that correspond to the zeros in the part of the array *Reg_N*(1, :) that corresponds to list(A) so that regions that no longer exist (the zeros) are no longer visited when searching the lists corresponding to region A. This function is called every time a region A merges with another region B. When a merger occurs, the list of neighbors of the new region A becomes the union of neighbors of A and B after deleting B from list(A), deleting A from list(B) and any common neighbors from list(B) to avoid duplications. Thus, at least two new zero entries exist in list(A) and must be skipped over by *Ptr_N*. Other zeros may exist in list(*A*) if a neighbor of *A* has merged with a region that is also a neighbor of *A* (see **Elim_Pnt**).

The first while loop in **close_ranks** modifies *Start_N(A)*, *End_N(A)*, and *Ptr_N* to skip over the zeros that might be at the beginning of list(*A*) in *Reg_N*(1, :). The second while loop in **close_ranks** modifies *End_N(A)*, and *Ptr_N* to skip over the zeros that might be in list(*A*) after some non-zero entries, interleaved between non-zero entries and/or at the end of list(*A*).

5.3.2.5 Function *New_Boundary*. The inputs to **New_Boundary** are *A*, *B*, *Start_B*, *End_B*, *Ptr_B*, and *Reg_B*. This function forms the boundary of the union of two regions *A* and *B* after the merger of *A* and *B*. There are two steps: (1) delete points that are common to both boundaries from the two lists of boundary points list(*A*) and list(*B*), and (2) concatenate the two lists and close ranks (skip over zero entries in *Reg_B*).

Step 1. Using a double while loop the function searches through list(*A*) and list(*B*) in *Reg_B* for common boundary points, deleting them by setting the corresponding entries in *Reg_B* to zero.

Step 2. The lists are concatenated using the following three statements as discussed in Section 5.1.1.

Ptr_B(End_B(A)) = Start_B(B)
End_B(A) = End_B(B)
Start_B(B) = 0.

A call to the function **close_ranks** modifies *Start_B*, *End_B* and the part of the array *Ptr_B* that corresponds to the list of new boundary points of *A*, to skip over the addresses in *Ptr_B* that correspond to zeros in the array *Reg_B* that resulted from the elimination of common boundary points.

5.3.3 Function *merge_pix*

The inputs to **merge_pix** are *A*, *B*, *Start*, *End*, and *Pointer*. This function simply concatenates the list of pixels of region *A* with the list of pixels of region *B* as discussed in Section 5.1.1 and Step 2 in the function **New_Boundary**.

5.3.4 Function *cls_rnks*

The inputs to **cls_rnks** are *Strt_S*, *End_S*, *P_S*, and *Start*. This function has the same structure and serves the same purpose (skip over zeros in *Start*) as the function **close_ranks** previously described. It modifies *Strt_S*, *End_S*, and *P_S* in order to skip over the zeros in the array *Start*.

When a merger occurs, the list of pixels of the new region *A* becomes the union of the pixels of *A* and the pixels of *B* and the array *Start* acquires a new zero at *Start(B)*. This function is called only after every multiple of *n* mergers has occurred.

5.4 PART IV. COLLECTING REGIONS AND BOUNDARIES AND THE PC-SEGMENTED IMAGE

$$\left\{\begin{array}{c} \text{Display\_Reg\_Pix} \\ \text{Vect\_to\_Img\_1} \\ \text{Display\_ith\_reg} \\ \text{Vect\_to\_Img\_1} \\ \text{Display\_Boundary} \\ \text{Vect\_to\_Img\_1} \\ \text{Display\_ith\_Bndry} \\ \text{Vect\_to\_Img\_1} \end{array}\right.$$

5.4.1 Function *Display_Reg_Pix*

The inputs to the function **Display_Reg_Pix** are *Start_S*, *Ptr_S*, *Start*, *Reg_Pix*, *nm*, and *C*. The outputs are *Regns*, *Segmentation*, and *Final_Regions*.

The arrays *Start_S* and *Ptr_S* are associated with the labels of the remaining regions (start and pointer). *Start_S* is the label of the first remaining region. *Ptr_S* is the pointer to subsequent regions. Using *Start_S* and *Ptr_S*, this function collects the labels of all the regions in the final segmentation and puts them in the array *Final_Regions*. The size of this array is then the number of regions left in the segmentation obtained.

The arrays *Start* and *Reg_Pix* are associated with the pixels in every remaining region (start and pointer). If A = *Start_S*, then A is the label of the first region and *Start(A)* is the first pixel in A. *Reg_Pix* is the pointer to subsequent pixels. Using *Start_S*, *Ptr_S*, *Start* and *Reg_Pix*, this function collects the pixels in every remaining region and forms the two row vectors *Segmentation* and *Regns* of dimension *nm*. If pixel x belongs to a region A, then

Regns(x) = A and *Segmentation(x)* = *C(A)* (single-channel), where *C(A)* is the PC value of the approximation to the original image that is associated with region *A*. Thus, after transforming the two row vectors *Segmentation* and *Regns* into an $n \times m$-matrix by the function **Vect_to_Img_1** in Section 5.4.2, these two matrices represent the remaining regions in the final segmentation (by label) and the piecewise approximation to the original image, respectively.

In the multichannel setting we simply replace the last scalar assignment statement above by a vector assignment:

Segmentation(:,x) = C(:,A) (multichannel),

where *C* is the $c \times nm$ array of piecewise linear coefficients. Then one must transform the $c \times nm$ array *Segmentation* into **c**-channel image data by, for example, applying **Vect_to_Img_1** to each of the rows of *Segmentation*.

5.4.2 Function *Vect_to_Img_1*

The function **Vect_to_Img_1** has inputs *m*, and *V*. The dimension of the vector *V* is a multiple of *m*, say *nm*. The function transforms the vector *V* into an $n \times m$-matrix called *Image*, the output. The components of the vector are laid along the rows of the matrix.

5.4.3 Function *Display_ith_reg*

The inputs to the function **Display_ith_reg** are *A*, *Start*, *Reg_Pix*, *nm*, and *C*. The output is the *nm* row vector *ith_region*. *A* is the label of the ith-region obtained using the array *Final_Regions* provided by the function **Display_Reg_Pix** of Section 5.4.1. This function does for region *A* what **Display_Reg_Pix** does for every region in the final segmentation; namely, it builds the *nm* row vector *ith_region*, which contains zeros everywhere except at the locations corresponding to the pixels of region *A*. At these locations it will either contain the label *A* or the constant value *C(A)* of the piecewise approximation to the original image that is associated with region *A* of the segmented image. The vector *ith_region* is then transformed into an array by the function **Vect_to_Img_1** that can be used to displays the ith-region by itself.

In the multichannel setting, we use the label *A* or *C(:,A)*, the ***c***-vector of coefficients. If the label *A* is used, then the output *ith_region* is a row vector that the function **Vect_to_Img_1** can transform into an $n{\times}m$ image. If the multichannel coefficient vector *C(:,A)* is used, then **Vect_to_Img_1** must be applied to each row of the $c{\times}nm$ array *ith_region*.

5.4.4 Function *Display_Boundary*

The inputs are *Final_Regions*, *Start_B*, *Ptr_B*, *Reg_B*, and *nm*. The outputs are *Bndrs* and *Reg_B*. The arrays *Start_B* and *Ptr_B* are the *start* and *pointer* arrays for *Reg_B*, which contains the boundary point labels. *Reg_B* will be modified by this function as described subsequently.

There are vertical virtual boundaries (see Section 5.1.2 about virtual boundary points) between consecutive pixels on the same row with labels 1 through *nm*. To display a vertical virtual boundary point, we use the pixel with the same label. Thus, this is a left representation in the sense that the boundary between two consecutive horizontal pixels is represented by the pixel on the left.

The horizontal virtual boundaries, labeled $(nm + 1)$ through $2nm$, are the boundaries between two consecutive pixels on the same column. To display a horizontal virtual boundary point, we use the pixel with label equal to the label of the boundary point minus *nm*. Thus, this is an up representation in the sense that the boundary between two consecutive vertical pixels is represented by the pixel on the top.

Remark. Giving the horizontal virtual boundaries a label different from the labels of the vertical ones is not necessary. At the end (at this point) we end up using the same pixel in the representation of the boundary points whose labels differ by *nm*. Thus, a different label is not necessary to begin with (see Section 5.1.2.).

This function builds the binary *nm*-row vector *Bndrs* with a one at every entry corresponding to a pixel representing a boundary point according to the preceding convention.

The array *Final_Regions* provides the labels of the regions in the final segmentation. If A = *Final_Regions(i)*, then A is the label of the ith-region. *Start_B(A)* is the address in *Reg_B* of the first boundary point of A; that is, *Reg_B(Start_B(A))* is the first boundary point of A. Using a while loop inside a for loop that sweeps through all the final regions, the function searches through the array of boundary labels in *Reg_B*. Only if *Reg_B(x)* > *nm*, is *Reg_B* modified by setting *Reg_B(x)* = *Reg_B(x)* - *nm*. Then, *Bndrs(Reg_B(x))* is set equal to 1.

5.4.5 Function *Display_ith_Bndry*

The function **Display_ith_Bndry** builds the binary *nm*-row vector *ith_Boundary*, the output of the function, which contains a one at every entry corresponding to a pixel representing a boundary point of the ith-region in the final segmentation according to the convention in the description of the previous function. It can be used to display the boundary of that region alone. The inputs to the function are *A*, *Start_B*, *Ptr_B*, *Reg_B*, and *nm*.

REFERENCES

1. D. Mumford and J. Shah. "Boundary Detection by Minimizing Functionals," *IEEE Conference on Computer Vision and Pattern Recognition*, San Francisco, Calif., 1985.

2. _______________. "Optimal Approximations by Piecewise Smooth Functions and Associated Variational Problems," *Commun. Pure Appl. Math*, Vol. XLII, No. 4 (1989).

3. G. Koepfler, C. Lopez, and L. Rudin. "Data Fusion by Segmentation. Application to Texture Discrimination," in *Proceedings of the 14th 'GRETSI Colloque', France, September 1993,* pp. 707-710.

4. L. Rudin, G. Koepfter, F. Nordby, and J. M. Morel. "Fast Variational Algorithm for Clutter Removal Through Pyramidal Domain Decomposition," in *SPIE Conference Proceedings, April 1994, Vol. 2037.*

5. Naval Air Warfare Center Weapons Division. *A General Merging Criterion*, by J. M. Martin. China Lake, Calif., NAWCWD, September 2002. 19 pp. (NAWCWD TP 8350, publication UNCLASSIFIED.)

6. __________. *Implementing the Region Growing Method Part 2: The Piecewise Affine Case*, by J. M. Martin. China Lake, Calif., NAWCWD, September 2002. 64 pp. (NAWCWD TP 8526, publication UNCLASSIFIED.)

Appendix
MATLAB PROGRAM

```
function [Regs,Segmt,Boundaries] = Tile_Max_Reg_Grow_B(n,m,level,Image,Threshold,lam)

                % PIECEWISE CONSTANT, ONE CHANNEL.
T=Threshold;
nm=n*m;
lambda=lam*lam*sqrt(nm);

'INITIALIZING'        % PART 1.
[S,E,RP,P_S,Strt_S,End_S]=initialize(nm);
[M,V,C,S_N,E_N,P_N,S_B,E_B,P_B,R_N,R_B]=NGB_PNTR_init_B(Image,lambda,n,m);

'TILING'          % PART 2.
[T1,T2]=tile(n,m,level);
[M,V,C,S_N,E_N,P_N,R_N,S,E,RP,S_B,E_B,P_B,R_B]=swp_rg_Tile_B(n,m,M,....
        V,C,T,T1,T2,level,S_N,E_N,P_N,R_N,S,E,RP,S_B,E_B,P_B,R_B);

'STEEPEST DESCENT'      % PART 3.
[R,N,MAX_Criter]=max_criterion(Strt_S,P_S,S_N,P_N,R_N);
iteration = -1;
while MAX_Criter > Threshold
  iteration=iteration+1;
  [M,V,C,S_N,E_N,P_N,R_N,S_B,E_B,P_B,R_B]=mrg_N_B(R,N,M,V,C,...
          S_N,E_N,P_N,R_N,S_B,E_B,P_B,R_B);
  [S,E,RP]=merge_pix(R,N,S,E,RP);
  if rem(iteration,fix(n)) == 0
    [Strt_S,End_S,P_S]=cls_rnks(Strt_S,End_S,P_S,S);
    ITERATION=iteration
  end
  [R,N,MAX_Criter]=max_criterion(Strt_S,P_S,S_N,P_N,R_N);
end
                % PART 4.
```

```
'COLLECTING REGIONS, BOUNDARIES, & THE PC-SEGMENTED IMAGE'
[Regs,Segmt,Final_Regions]=Display_Reg_Pix(Strt_S,P_S,S,RP,C);
Number_of_Regions=length(Final_Regions)
Regs=Vect_to_Img_1(m,Regs);
Segmt=Vect_to_Img_1(m,Segmt);      % SINGLE-CHANNEL
%for i=1:length(Final_Regions)
%  ith_region=Display_ith_reg(Final_Regions(i),S,RP,nm,C);
%  ith_region=Vect_to_Img_1(m,ith_region);
%  pause
%end
[Boundaries,R_B]=Display_Boundaries(Final_Regions,S_B,P_B,R_B,nm);
Boundaries=Vect_to_Img_1(m,Boundaries);
%for i=1:length(Final_Regions)
%  ith_Boundary=Displ_ith_Bndry(Final_Regions(i),S_B,P_B,R_B,nm);
%  ith_Boundary=Vect_to_Img_1(m,ith_Boundary);
%  pause
%end
subplot(3,1,2),imagesc(Segmt); axis image;      %mesh(Segmt)
subplot(3,1,3),imagesc(Boundaries); axis image;   %mesh(Boundaries)
```

PART I
INITIALIZING

```
function [Start,End,Reg_Pix,P_S,Strt_S,End_S] = initialize(nm)

% To initialize Start, End, Reg_Pix, P_S, Strt_S, and End_S

Reg_Pix=zeros(1,nm);  % Pointer to pixels in region k. Pixels start at
Start=[1:nm];         % Start(k) and continue along Reg_Pix(*) by
End=Start;            % iterating: Reg_Pix(Start(k)),
                      % Reg_Pix(Reg_Pix(Start(k))),...untill we reach
                      % a zero; i.e. Reg_Pix(End(k))=0.
P_S=[[2:nm] 0];       % Pointer to the non-zero entries in Start.
Strt_S=1;             % Beginning of non-zero entries in Start.
End_S=nm;             % End of non-zero entries in Start.
```

```
function [M,V,C,Start_N,End_N,Ptr_N,Start_B,End_B,...
Ptr_B,Reg_N,Reg_B] = NGB_PNTR_init_B(Image,lambda,n,m)

% To initialize M, V, C, Start_N, End_N, Ptr_N, Start_B, End_B, Ptr_B,
% Reg_N, and Reg_B.

% Reg_N(1,:)=List of neighbors.
% Reg_N(2,:)=List of common boundary lengths.
% Reg_N(3,:)=List of merging criteria.

% Reg_B=List of common boundary points.

% Start_N=beginnings of neighbor lists.
% Start_B=beginnings of boundary points lists.

% End_N=ends of neighbor lists.
% End_B=ends of boundary points lists.

% Ptr_N=pointer to neighbors in the lists in Reg_N(1,:), a 1x4nm-array.
% Ptr_B=pointer to boundary points in the lists in Reg_N(4,:).

% Initially Start_N=Start_B, End_N=End_B, and Ptr_N=Ptr_B.

nm=n*m;
Q=ones(1,nm);
M=Q;
V=Img_Vect_1(n,m,Image);    % For multichannel data use:
                    % for i=1:ch
                    %     V(i,:)=Img_Vect_1(n,m,CHANNEL(i));
                    % end
C=V;

End_N=4*[1:nm];
Start_N=End_N-3*Q;

Ptr_N(End_N)=zeros(1,nm);
Ptr_N(Start_N)=Start_N+Q;
Ptr_N(Start_N+Q)=Start_N+2*Q;
Ptr_N(Start_N+2*Q)=Start_N+3*Q;

Start_B=Start_N;
```

```
End_B=End_N;
Ptr_B=Ptr_N;

Reg_N=zeros(3,4*nm);
Reg_B=zeros(1,4*nm);

for i=2:n-1
  for j=2:m-1    % u, l, d, r.
    k=(i-1)*m+j;
    k4=4*k;
    Reg_N(1,k4-3)=k-m;    % u(k)
    Reg_N(2,k4-3)=1;
    Reg_N(1,k4-2)=k-1;    % l(k)
    Reg_N(2,k4-2)=1;
    Reg_N(1,k4-1)=k+m;    % d(k)
    Reg_N(2,k4-1)=1;
    Reg_N(1,k4)=k+1;      % r(k)
    Reg_N(2,k4)=1;

    Reg_B(k4-3)=nm+k-m;   % u(k)
    Reg_B(k4-2)=k-1;      % l(k)
    Reg_B(k4-1)=nm+k;     % d(k)
    Reg_B(k4)=k;          % r(k)
  end
  % j=m        u, l, d.
    k=i*m;
    k4=4*k;
    Reg_N(1,k4-3)=k-m;    % u(k)
    Reg_N(2,k4-3)=1;
    Reg_N(1,k4-2)=k-1;    % l(k)
    Reg_N(2,k4-2)=1;
    Reg_N(1,k4-1)=k+m;    % d(k)
    Reg_N(2,k4-1)=1;

    Reg_B(k4-3)=nm+k-m;   % u(k)
    Reg_B(k4-2)=k-1;      % l(k)
    Reg_B(k4-1)=nm+k;     % d(k)

  % j=1        u, d, r.
    k=(i-1)*m+1;
    k4=4*k;
    Reg_N(1,k4-3)=k-m;    % u(k)
    Reg_N(2,k4-3)=1;
    Reg_N(1,k4-1)=k+m;    % d(k)
    Reg_N(2,k4-1)=1;
    Reg_N(1,k4)=k+1;      % r(k)
    Reg_N(2,k4)=1;

    Reg_B(k4-3)=nm+k-m;   % u(k)
    Reg_B(k4-1)=nm+k;     % d(k)
    Reg_B(k4)=k;          % r(k)

end
%  i=1        l, d, r.
for j=2:m-1
  k=j;
  k4=4*k;
  Reg_N(1,k4-2)=k-1;    % l(k)
  Reg_N(2,k4-2)=1;
  Reg_N(1,k4-1)=k+m;    % d(k)
  Reg_N(2,k4-1)=1;
  Reg_N(1,k4)=k+1;      % r(k)
  Reg_N(2,k4)=1;

  Reg_B(k4-2)=k-1;      % l(k)
  Reg_B(k4-1)=nm+k;     % d(k)
  Reg_B(k4)=k;          % r(k)

%  i=n        u, l, r.
  k=(n-1)*m+j;
    k4=4*k;
    Reg_N(1,k4-3)=k-m;    % u(k)
    Reg_N(2,k4-3)=1;
    Reg_N(1,k4-2)=k-1;    % l(k)
    Reg_N(2,k4-2)=1;
    Reg_N(1,k4)=k+1;      % r(k)
    Reg_N(2,k4)=1;

    Reg_B(k4-3)=nm+k-m;   % u(k)
    Reg_B(k4-2)=k-1;      % l(k)
    Reg_B(k4)=k;          % r(k)

 end
 % (i,j)=(1,1), k=1,   d, r.

  Reg_N(1,3)=m+1;      % d(k)
  Reg_N(2,3)=1;
  Reg_N(1,4)=2;        % r(k)
  Reg_N(2,4)=1;

  Reg_B(3)=nm+1;       % d(k)
  Reg_B(4)=1;          % r(k)

 % (i,j)=(1,m), k=m,   l, d.

  k4=4*m;
  Reg_N(1,k4-2)=m-1;    % l(k)
  Reg_N(2,k4-2)=1;
  Reg_N(1,k4-1)=m+m;    % d(k)
  Reg_N(2,k4-1)=1;

  Reg_B(k4-2)=m-1;      % l(k)
  Reg_B(k4-1)=nm+m;     % d(k)

 % (i,j)=(n,1)    u, r.

  k=(n-1)*m+1;
  k4=4*k;
  Reg_N(1,k4-3)=k-m;    % u(k)
  Reg_N(2,k4-3)=1;
  Reg_N(1,k4)=k+1;      % r(k)
  Reg_N(2,k4)=1;

  Reg_B(k4-3)=nm+k-m;   % u(k)
  Reg_B(k4)=k;          % r(k)

 % (i,j)=(n,m)    u, l.

  k=nm;
  k4=4*k;
```

```
Reg_N(1,k4-3)=k-m;    % u(k)
Reg_N(2,k4-3)=1;
Reg_N(1,k4-2)=k-1;    % l(k)
Reg_N(2,k4-2)=1;

Reg_B(k4-3)=nm+k-m;    % u(k)
Reg_B(k4-2)=k-1;       % l(k)

Reg_N(2,:)=lambda*Reg_N(2,:);

for k=1:nm   % To compute the Merging Criterion(A,B) for all A=1,2,...,nm
         % and all neighbors B of A. A=k and B=Reg_N(1,Ad(i)).
  [Ad,N,L,Crit]=fetch_ADD_NGB(k,Start_N,Ptr_N,Reg_N);
  if length(Ad) >= 1
    len=length(Ad);
    for i=1:len
      M_AB=M(k)+M(Reg_N(1,Ad(i))); % B=Reg_N(1,Ad(i)) is a nghbr of A.
      M_inv_AB=1/M_AB;
      H=M(k)*M_inv_AB*M(Reg_N(1,Ad(i)));
      Reg_N(3,Ad(i))=lambda-(C(k)-C(Reg_N(1,Ad(i))))'*H*(C(k)-C(Reg_N(1,Ad(i))));
      % FOR MULTICHANNEL DATA USE
      %Reg_N(3,Ad(i))=lambda-(C(:,k)-C(:,Reg_N(1,Ad(i))))'*H*(C(:,k)-C(:,Reg_N(1,Ad(i))));
    end
  end
end
```

```
function V = Img_Vect_1(n,m,Image)

% This function transforms a matrix representation of an image to a vector
% representation of the image by concatenating the rows of the nxm-matrix
% to form an nm-row-vector V.
% Jorge M. Martin, NAWC-WPNS Code 474400D, China Lake, CA. April 1996.

for i=1:n
  i_1=i-1;
  for j=1:m
    V(1,i_1*m+j)=Image(i,j);
  end
end
```

```
function [Addresses,Neighbs,Lengths,Criter] = fetch_ADD_NGB(A,Start_N,Ptr_N,Reg_N)

% To get the addresses, labels, neighbors, boundary lengths,
% and criteria of region A.
% Addresses are the addresses of the labels of the neighbors of A.
% Neighbs are the labels of the neighbors of A.

if Start_N(A) == 0
  'A has no neihgbors, A has been absorbed.'
  return
end

x=Start_N(A);
while x ~= 0
  if Reg_N(1,x) ~= 0                % First remaining neighbor of A.
     Addresses=[x];                 % The address of the first neighbor.
     Neighbs=[Reg_N(1,x)];          % The first neighbor.
     Lengths=[Reg_N(2,x)];          % The first bouncary length.
     Criter=[Reg_N(3,x)];           % The first merging criterion.
  end
  x=Ptr_N(x);                       % Next neighbor.
end
```

PART II
TILING

```
function [T1,T2] = tile(n,m,level)

% To define the centers of the first four level tiles.
% level=1,2,3 or 4.

T1=sweep_array_New(n,m);
if level <= 2
  T2=[];
  return
end
```

```
T2=sweep_array_4(n,m,T1(2,:));
```

```
function Swp_Arr = sweep_array_New(n,m)

% To define the centers of the first two level tiles.
% Swp_Arr=sweep_array_New(18,12)

a=zeros(1,5);

a(1)=fix(m/5);
a(2)=fix((m+3)/5);
a(3)=fix((m+1)/5);
a(4)=fix((m+4)/5);
a(5)=fix((m+2)/5);

b=fix(n/5);
c=rem(n,5);

dim=b*sum(a);
if c >= 1
  dim=dim+sum(a(1:c));
end
Swp_Arr=zeros(2,dim);

Swp_Arr(1,1:a(1))=5*[1:a(1)];

aa=a(1)+a(2);
Swp_Arr(1,a(1)+1:aa)=(m-3)+5*[1:a(2)];

aaa=aa+a(3);
Swp_Arr(1,aa+1:aaa)=(2*m-1)+5*[1:a(3)];

if n >= 3
  Swp_Arr(2,1:a(3))=Swp_Arr(1,aa+1:aaa);  % First block of Swp_Arr(2,:).
end

aa=aaa;
aaa=aa+a(4);
Swp_Arr(1,aa+1:aaa)=(3*m-4)+5*[1:a(4)];
```

```
aa=aaa;
aaa=aa+a(5);
Swp_Arr(1,aa+1:aaa)=(4*m-2)+5*[1:a(5)];

if b > 1
  for i=1:b-1
    Swp_Arr(1,i*aaa+1:(i+1)*aaa)=5*m+Swp_Arr(1,(i-1)*aaa+1:i*aaa);
  end
end

if c >= 1
  aa=b*sum(a);
  aaa=aa+sum(a(1:c));
  Swp_Arr(1,aa+1:aaa)=b*5*m+Swp_Arr(1,1:aaa-aa);
end

%  The rest of Swp_Arr(2,:).

if n >= 8
  d=fix((n-3)/5);    % d >= 1.
  for i=1:d
    Swp_Arr(2,i*a(3)+1:(i+1)*a(3))=(2+5*i)*m-1+5*[1:a(3)];  % a(3)-Blocks.
  end
end
```

```
function Swp_Arr = sweep_array_4(N,M,Arr)

% To define the centers of the level 3 and 4 tiles.

m=fix((M+1)/5);
n=1+fix((N-3)/5);
Array=Arr(1:n*m);  %  <====== Not necessary ??.
a=zeros(1,5);

a(1)=fix(m/5);
a(2)=fix((m+3)/5);
a(3)=fix((m+1)/5);
a(4)=fix((m+4)/5);
```

```
a(5)=fix((m+2)/5);

b=fix(n/5);
c=rem(n,5);

if a(1) == 0
  return
end

dim=b*sum(a);
if c >= 1
  dim=dim+sum(a(1:c));
end
Swp_Arr=zeros(2,dim);

Swp_Arr(1,1:a(1))=Array(5*[1:a(1)]);

aa=a(1)+a(2);
Swp_Arr(1,a(1)+1:aa)=Array((m-3)+5*[1:a(2)]);

aaa=aa+a(3);
Swp_Arr(1,aa+1:aaa)=Array((2*m-1)+5*[1:a(3)]);

if n >= 3
  Swp_Arr(2,1:a(3))=Swp_Arr(1,aa+1:aaa);  % First block of Swp_Arr(2,:).
end

aa=aaa;
aaa=aa+a(4);
Swp_Arr(1,aa+1:aaa)=Array((3*m-4)+5*[1:a(4)]);

aa=aaa;
aaa=aa+a(5);
Swp_Arr(1,aa+1:aaa)=Array((4*m-2)+5*[1:a(5)]);

if b > 1
  for i=1:b-1
    Swp_Arr(1,i*aaa+1:(i+1)*aaa)=25*M+Swp_Arr(1,(i-1)*aaa+1:i*aaa);
  end
end
```

```
if c >= 1
  aa=b*sum(a);
  aaa=aa+sum(a(1:c));
  Swp_Arr(1,aa+1:aaa)=b*25*M+Swp_Arr(1,1:aaa-aa);
end

% The rest of Swp_Arr(2,:).

if n >= 8
  d=fix((n-3)/5);    % d >= 1.
  for i=1:d
    Swp_Arr(2,i*a(3)+1:(i+1)*a(3))=25*i*M+Swp_Arr(2,1:a(3));   % a(3)-Blocks.
  end
end
```

```
function [M,V,C,S_N,E_N,P_N,R_N,S,E,RP,S_B,E_B,P_B,R_B] = swp_rg_Tile_B(n,m,M,....
               V,C,T,T1,T2,level,S_N,E_N,P_N,R_N,S,E,RP,S_B,E_B,P_B,R_B)

% level=1,2 3, or 4.

nm=n*m;
J=1
if level >= 2
  J=2;
end
for j=1:J
for k=1:length(T1(1,:))
  A=T1(j,k);                 % Region A.
  if A ~= 0
  if S_N(A) ~= 0             % Region A has not been absorbed yet.
    [Addresses,Neighbs,Lengths,Criter]=fetch_ADD_NGB(A,S_N,P_N,R_N);
    L=length(Neighbs);
    if L >= 1
      for i=1:L
        B=Neighbs(i);
        if Criter(i) > T     % Merge A and B=Neighbs(i).
          [M,V,C,S_N,E_N,P_N,R_N,S_B,E_B,P_B,R_B]=mrg_N_B(A,B,M,V,C,...
                   S_N,E_N,P_N,R_N,S_B,E_B,P_B,R_B);
          [S,E,RP]=merge_pix(A,B,S,E,RP); % RP is a pointer.
        end
      end
    end
  end
  end
end
end

if level >= 3
J=1;
if level >= 4
  J=2
end
for j=1:J
for k=1:length(T2(1,:))
  A=T2(j,k);                    % Region A.
  if A ~= 0
  if S_N(A) ~= 0                % Region A has not been absorbed yet.
    [Addresses,Neighbs,Lengths,Criter]=fetch_ADD_NGB(A,S_N,P_N,R_N);
    L=length(Neighbs);
    if L >= 1
      for i=1:L
        B=Neighbs(i);
        if Criter(i) > T        % Merge A and B=Neighbs(i).
          [M,V,C,S_N,E_N,P_N,R_N,S_B,E_B,P_B,R_B]=mrg_N_B(A,B,M,V,C,...
                      S_N,E_N,P_N,R_N,S_B,E_B,P_B,R_B);
          [S,E,RP]=merge_pix(A,B,S,E,RP);  % RP is a pointer.
        end
      end
    end
  end
  end
end
end
end
```

PART III
STEEPEST DESCENT

```
function [Region,Neighbor,MAX_Criter] = max_criterion(Start_S,Ptr_S,Start_N,...
                                        Ptr_N,Reg_N)

% To find the max criterion and the region and neighbor corresponding to it

Region=0;
Neighbor=0;
MAX_Criter=-1;

k=Start_S;                          % First non-0 region. Replaces "for k=1:nm"
while k ~= 0
   x=Start_N(k);                    % First neignbor of region k.
   while x ~= 0
      if Reg_N(1,x) ~= 0            % A non zero neighbor.
        if MAX_Criter < Reg_N(3,x)  % Merging criterion between k & x.
          MAX_Criter=Reg_N(3,x);    % Merging criterion is larger.
          Neighbor=Reg_N(1,x);      % Neigbor with largest citerion yet.
          Region=k;                 % Region with largest citerion yet.
        end
      end
      x=Ptr_N(x);                   % Next neighbor.
   end
   k=Ptr_S(k);                      % Next region k until k=0.
end
```

```
function [M,V,C,Start_N,End_N,Ptr_N,Reg_N,Start_B,End_B,Ptr_B,Reg_B] = mrg_N_B(A,...
                 B,M,V,C,Start_N,End_N,Ptr_N,Reg_N,Start_B,End_B,Ptr_B,Reg_B)

% To update the pointer "Ptr_N" and the arrays Start_N, End_N, and "Reg_N"
% when the merger of two regions A and B occurs.
% The pointer points to "Reg_N" which contains the labels of the neighbors
% of each region. When regions A and B are merged, the new region formed is
% labeled region A. The addresses of the neighbors of A start at
% "Start_N(A)"and end at "End_N(A)". They are: Start_N(A),
% Ptr_N(Start_N(A)), Ptr_N(Ptr_N(Start_N(A))), ...; Ptr_N(End_N(A))=0 marks
% the end of the list for A.
```

```
%
% C is a common neighbor of A and B which are to be merged; A will absorb B
% To eliminate B as a neighbor from the list(C), eliminate C from the
% list(B) to avoid duplication of C in the merged list(AUB), and provide a
% pointer to A in the list(C) to increment length(A,C) in list(C) after the
% merging of A and B.
% ..............................................................................................................................

%                    PART 1. PC-2D-1Ch. and % Multichannel Changes.

% Update M(A), V(A), and C(A).

M(A)=M(A)+M(B);            % a scalar.                Single & Multichannel.
M_inv_A=1/M(A);            % a scalar.                Single & Multichannel.
V(A)=V(A)+V(B);            % V(:,A)=V(:,A)+V(:,B);    Multichannel.
C(A)=M_inv_A*V(A);         % C(:,A)=M_inv_A*V(:,A);   Multichannel.
%..............................................................................................................................

%                        PART 2.

x=Start_N(A);
y=Start_N(B);
while x ~= 0
  while y ~= 0
    if Reg_N(1,x) ~= 0 & Reg_N(1,x) == Reg_N(1,y)
                                   % Found a common neighbor CN = Reg_N(1,x).
      CN=Reg_N(1,x);               % x = pointer to CN in list(A)
                                   % y = pointer to CN in list(B)
      Reg_N(1,y)=0;                % Delete CN from list(B) to avoid
                                   % duplication in the merged list(AUB).

      [Pnt_to_A_inC,Reg_N]=Elim_Pnt(Reg_N(1,x),B,A,Start_N,Ptr_N,Reg_N);

                                   % Eliminate B from list(C); (B will be absorbed
                                   % by A), and provide pointer_to_A_in_C to update
                                   % length(A,C) and Criterion(A,C) in list(C).

      Reg_N(2,x)=Reg_N(2,x)+Reg_N(2,y);       % Update length(A,C) in list(A):
                                              % length(A,C)=length(A,C)+length(B,C).
      Reg_N(2,Pnt_to_A_inC)=Reg_N(2,x);       % Update length(A,C) in list(C):
```

```
                                                    % length(A,C)=length(A,C)+length(B,C).
        Reg_N=Criterion(A,CN,x,M,C,Reg_N);          % New merging criterion
                                                    % between A and CN put in list(A).
        Reg_N(3,Pnt_to_A_inC)=Reg_N(3,x);           % New merging criterion
                                                    % between A and C put in list(C).

      end % if
      y=Ptr_N(y);
    end % while y ~= 0
    x=Ptr_N(x);
    y=Start_N(B);
  end % while x~= 0
%.............................................................................................

%                       PART 3.

Reg_N=Elim_and_Update(A,B,C,M,Start_N,Ptr_N,Reg_N);

                                                    % Update list(A), list(B), and
                                                    % the lists of the members of list(B)
                                                    % before concatenating the two lists.
%.............................................................................................

%                       PART 4.

Ptr_N(End_N(A))=Start_N(B);                   % To concatenate the lists for A and B.
End_N(A)=End_N(B);                            % The end of the list for A is now at the end of B.
Start_N(B)=0;                                 % Region B has been absorbed by A.

[Start_N,End_N,Ptr_N]=close_ranks(A,Start_N,End_N,Ptr_N,Reg_N);
[Start_B,End_B,Ptr_B,Reg_B]=New_Boundary(A,B,Start_B,End_B,Ptr_B,Reg_B);
```

```
function [Pnt_to_A_inC,Reg_N] = Elim_Pnt(C,B,A,Start_N,Ptr_N,Reg_N)

% Partially updating the list(C).
% C is a common neighbor of A and B which are to be merged; A will absorb B.

% (1) To eliminate B as a neighbor from the list(C).
% (2) To provide a pointer to A in the list(C) to update length(A,C) and
%     criterion(A,C) in list(C) after the merging of A and B.
```

```
  x=Start_N(C);
  while x ~= 0
    if Reg_N(1,x) == B
      Reg_N(1,x)=0;                  % Region B is no longer a neighbor of C.
    elseif Reg_N(1,x) == A
      Pnt_to_A_inC=x;                % Pointer to A in the list(C).
    end
    x=Ptr_N(x);                      % Check next neighbor to see if it is B.
  end
```

```
function Reg_N = Criterion(A,C,pointer,M,Coeff,Reg_N)

% Already Multichannel.
% To update the criterion(A,C) in list(A).
% pointer = pointer to C in list(A).

M_AC=M(A)+M(C);
M_inv=1/M_AC;
H=M(A)*M_inv*M(C);
Reg_N(3,pointer)=Reg_N(2,pointer)-(Coeff(A)-Coeff(C))'*H*(Coeff(A)-Coeff(C));
```

```
function Reg_N = Elim_and_Update(A,B,Coeff,M,Start_N,Ptr_N,Reg_N)

% After taking care of common neighbors and before merging the lists of A -
% and B (B will be absorbed by A.), we need to update list(A) and list(B)
% as follows.

% Updating list(A):

% (1) Eliminate B as neighbor from the list(A).
% (2) Update criterion(A,C) for every neighbor C in list(A) exept neighb B.
% (3) Update criterion(A,C) in list(C) for every neighbor C in list(A)
%     exept neighbor B.

% Updating list(B) and the lists of the members of list(B):

% (1) Eliminate A as neighbor from the list(B).
% (2) Update criterion(A,C) for every neighbor C in list(B) exept neighb A.
% (3) Replace B by A in the list(C) for all the neighbors C of B in list(B)
%     except neighbor A and update criterion(A,C) in list(C).

% NOTE that if C is a common neighbor of A and B, then B will have been
% replaced by 0 at the common neighbor stage (i.e. function "Elim_Pnt").
% Thus there will be no duplication of A in the list(C).
%.......................................................................

% Updating list(A).

x=Start_N(A);              % x=pointer to first neighbor C=Reg_N(1,x) in list(A).
while x ~= 0               % Ptr_N(x)=pnt to next nghb C=Reg_N(1,Ptr_N(x)) in list(A).
  if Reg_N(1,x) == B
    Reg_N(1,x)=0;                                                       %Step (1).
  elseif Reg_N(1,x) ~= 0
    Reg_N=Criterion(A,Reg_N(1,x),x,M,Coeff,Reg_N);                      %Step (2).
    Reg_N=Update_criterion(Reg_N(1,x),A,M,Coeff,Start_N,Ptr_N,Reg_N);   %Step (3).
  end
  x=Ptr_N(x);              % Check next neighbor to see if it is B.
end
%.......................................................................

% Updating list(B) and the lists of the members of list(B).

x=Start_N(B);              % x=pointer to first neighbor C=Reg_N(1,x) in list(B).
while x ~= 0
  if Reg_N(1,x) == A
    Reg_N(1,x)=0;                                                       % Step (1).
  elseif Reg_N(1,x) ~= 0
    Reg_N=Criterion(A,Reg_N(1,x),x,M,Coeff,Reg_N);                      % Step (2).
    Reg_N=replaceB_by_A(A,B,Reg_N(1,x),Coeff,M,Start_N,Ptr_N,Reg_N);    % Step (3).
  end
  x=Ptr_N(x);              % Check next neighbor to see if it is A.
end
```

```
function [Start_N,End_N,Ptr_N] = close_ranks(A,Start_N,End_N,Ptr_N,Reg_N)

% To skip over the addresses in the pointer Ptr_N that correspond to
% regions in list(A) that have been absorbed and adjust Start_N and End_N.

% Note that when region "A" absorbs region "B" the list for "A" becomes the
% union of list(A) and list(B), and it has at least two new zeros; one for
% "B" in list(A) and one for "A" in list(B). Thus, the array Reg_N(1,:) has
% at least two new zeros as well.
% The lables of the remaining regions are the non-zero entries
% of Reg_N(1,:).
% Here, region "A" has absorbed region "B"; however, we check all zeros in
% list(A). Extra zeros may exist in list(A) if a neighbor of "A" has
% absorbed a region that is also a neighbor of "A" (see "Elim_Pnt").

x=Start_N(A);
if x ~= 0
  End_N(A)=x;
  while Reg_N(1,x) == 0
    x=Ptr_N(x);
    if x == 0
      return
    end
    Start_N(A)=x;
    End_N(A)=x;
  end

  while Ptr_N(x) ~= 0
    y=Ptr_N(x);
    if y == 0
      return
    end
    while Reg_N(1,y) == 0
      Ptr_N(x)=Ptr_N(y);              % Zeros will be "skipped" by Ptr_N in list(A).
      y=Ptr_N(y);
      if y == 0
        return
      end
    end
    x=y;                              % Continue looking for zeros until Ptr_N(x)=0.
    End_N(A)=y;
  end
end
```

```
function [Start_B,End_B,Ptr_B,Reg_B] = New_Boundary(A,B,Start_B,End_B,Ptr_B,Reg_B)

% To obtain the boundary of the union of the two regions A and B.
% The addresses of the boundary points of A are Start_B(A), Ptr_B(Start_B(A)),
% ...Ptr_B(...Ptr_B(Start_B(A))), ...End_B(A); Ptr_B(End_B(A))=0 marks the end
% of the list. Same for B.

% There are two steps:  (1) Delete points that are common to both boundaries
% and                   (2) Concatenate the two lists and close ranks..

% Step 1.

x=Start_B(A);
y=Start_B(B);
while x ~= 0
  while y ~= 0
    if Reg_B(x) == Reg_B(y)
      Reg_B(x)=0;
      Reg_B(y)=0;                        % and/or close ranks!
    end
    y=Ptr_B(y);
  end
  x=Ptr_B(x);
  y=Start_B(B);
end

% Step 2.

Ptr_B(End_B(A))=Start_B(B);          % To concatenate the lists for A and B.
End_B(A)=End_B(B);                   % The end of the list for A is now at the end of B.
Start_B(B)=0;                        % Region B has been absorbed by A.

[Start_B,End_B,Ptr_B]=close_ranks(A,Start_B,End_B,Ptr_B,Reg_B);
```

```
function Reg_N = Update_criterion(C,A,M,Coeff,Start_N,Ptr_N,Reg_N)

% To update Criterion(C,A) in list(C).
```

```
x=Start_N(C);                                % x = pointer to the first neighbor of C.
while x ~= 0
  if Reg_N(1,x) == A                         % Update Reg_N(3,x)=criterion(C,A).
    Reg_N=Criterion(C,A,x,M,Coeff,Reg_N);
  end
  x=Ptr_N(x);                                % Check next neighbor to see if it is A.
end
```

---

function Reg_N = **replaceB_by_A**(A,B,C,Coeff,M,Start_N,Ptr_N,Reg_N)

```
% To replase B by A in list(C) and update criterion(A,C) in list(C).

x=Start_N(C);                              % x = pointer to the first neighbor of C.
while x ~= 0
  if Reg_N(1,x) == B                       % Replace B by A and update criterion(A,C) in
                                           % list(C).
    Reg_N(1,x)=A;                          % Region B has been replaced by A.
    Reg_N=Criterion(C,A,x,M,Coeff,Reg_N);
  end
  x=Ptr_N(x);                              % Check next neighbor to see if it is B.
end
```

---

function [Start,End,Pointer] = **merge_pix**(A,B,Start,End,Pointer)

```
% To update the pointer "Pointer" when a merging of regions A and B occurs.
% The pointer points to the pixels in regions A and B. The new region is
% labled A.

Pointer(End(A))=Start(B);
End(A)=End(B);
Start(B)=0;
```

---

function [Strt_S,End_S,P_S] = **cls_rnks**(Strt_S,End_S,P_S,Start)

```
% To skip over the addresses in the pointer P_S that correspond to
% a region that has been absorbed and adjust Strt_S and End_S.
% Note that when region "A" absorbs region "B" the pixels for "A" becomes
% the union of pixels(A) and pixels(B), and Start has a new zero at
% Start(B).

 x=Strt_S;
 End_S=x;
 while Start(x) == 0
   x=P_S(x);
   if x == 0
     return
   end
   Strt_S=x;
   End_S=x;
 end

 while P_S(x) ~= 0
   y=P_S(x);
   if y == 0, return, end
   while Start(y) == 0
     P_S(x)=P_S(y);              % A zero will be "skipped" by P_S.
     y=P_S(y);
     if y == 0
       return
     end
   end
   x=y;                          % Check next entry.
   End_S=y;
 end
```

PART IV
COLLECTING REGIONS & BOUNDARIES & THE PC-SEGMENTED IMAGE

```
function [Regns,Segmentation,Final_Regions] = Display_Reg_Pix(Start_S,Ptr_S,Start,...
                                                  Reg_Pix,C)

% To collect the lables of all the regions in the final segmentation and
% put them in the array Final_Regions. The size of Final_Regions is the
% number of regions in the segmentation.
% To assign the lable "A" to every pixel in region A for every region A in
% the final segmentation and put it in the 1xnm-array Regns.

% To assign the piecewise constant value C(A) to every pixel in region A
% for every region A in the final segmentation and put it in the 1xnm-array
% Segmentation.

% Start_S and Ptr_S are associated with the lables of regions .
% Start and Reg_Pix are associated with the pixels ("start & pointer").

Final_Regions=[];

A=Start_S;                              % First region in the final segmetation has lable A.
while A ~= 0
   if Start(A) ~= 0
      Final_Regions=[Final_Regions A];  % A is a final region.
      x=Start(A);                       % First pixel in region A.
      Regns(x)=A;                       % Pixel x receives lable A.
      Segmentation(x)=C(A);          % In the segmented image, pixel x has PC-value C(A).
      % Segmentation(:,x)=C(:,A);       % MULTICHANNEL PC-value.
      x=Reg_Pix(x);                     % Next pixel in region A.
      while x ~= 0
         Regns(x)=A;                    % All pixels x in A receive lable A.
         Segmentation(x)=C(A);          % All pixels x in A have PC-value C(A).
         x=Reg_Pix(x);                  % Next pixel in region A.
      end
   end
   A=Ptr_S(A);                          % Next region in the final segmentation.
end
```

```
function Image = Vect_to_Img_1(m,V)

% This function transforms a vector representation of an image V into a
% matrix representation "Image". The matrix "Image" is nxm.
% Jorge M. Martin, NAWC-WPNS Code 474400D, China Lake, CA. April 1996.

L=length(V);               % L=nm should be a multiple of m.
for k=1:L
```

```
  Image(fix((k-1)/m)+1,rem(k-1,m)+1)=V(k);
end
```

---

function ith_region = **Display_ith_reg**(A,Start,Reg_Pix,nm,C)

```
% A is the lable of the ith-region. The nm-row vector "ith_region" will
% contain zeros everywhere exept at locations corresponding to the pixels
% of region A. At these locations it will either contain the lable "A" or
% the constant value C(A) of the approximation to the original image that
% is associated with region A of the segmented image.

ith_region=zeros(1,nm);

  if Start(A) ~= 0
    x=Start(A);                       % x is the First pixel in region A.
    ith_region(x)=A;                  % Lable A put at location x...,or...
  %  ith_region(x)=C(A);              % constant value C(A) put at location x.
    x=Reg_Pix(x);                     % Next pixel in region A.
    while x ~= 0
       ith_region(x)=A;               % Lable A put at location x...,or...
    %   ith_region(x)=C(A);           % constant value C(A) put at location x.
       x=Reg_Pix(x);                  % Next pixel in region A.
    end
  end
```

---

function [Bndrs,Reg_B] = **Display_Boundaries**(Final_Regions,Start_B,Ptr_B,Reg_B,nm)

```
% To put a one on every k in {1,2,...,nm} that is a boundary point.

Bndrs=zeros(1,nm);
N=length(Final_Regions);
for i=1:N
  x=Start_B(Final_Regions(i));
  while x ~= 0
    if Reg_B(x) > nm
      Reg_B(x)=Reg_B(x)-nm;
    end
    if Reg_B(x) ~= 0
```

```
      Bndrs(Reg_B(x))=1;
    end
    x=Ptr_B(x);
  end
end
```

```
function ith_Boundary = Displ_ith_Bndry(A,Start_B,Ptr_B,Reg_B,nm)

% To get the pixels of the ith-boundary.

ith_Boundary=zeros(1,nm);
x=Start_B(A);
while x ~= 0
  if Reg_B(x) ~= 0
    ith_Boundary(Reg_B(x))=1;
  end
  x=Ptr_B(x);
end
```

INITIAL DISTRIBUTION

1 Naval War College, Newport (1E22, President)
1 Headquarters, 497 INOT, Falls Church (IOG Representative)
2 Defense Technical Information Center, Fort Belvoir
1 Center for Naval Analyses, Alexandria, VA (Technical Library)

ON-SITE DISTRIBUTION

3 Code 4TL000D (2 plus Archives copy)
22 Code 4T4100D
    S. Chesnut (1)
    J. Martin (21)